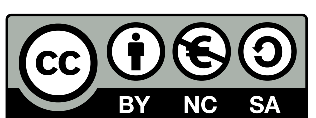
UV élective

Languages & logic

ELU 610

Responsables : J-C. Bach, É. Cousin, F. Dagnat and Y. Haralambous

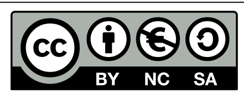
1st semester 2019



Authors

Jean-Christophe Bach, Antoine Beugnard, Éric Cousin, Fabien Dagnat, Yannis Haralambous

Ce support de cours est diffusé sous licence Creative Commons BY NC SA. (http://creativecommons.org/licenses/by-nc-sa/2.0/fr/)



Vous êtes libres de :

— reproduire, distribuer et communiquer cette création au public ;
— modifier cette création

selon les conditions suivantes :

— Paternité (**BY**) : vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'oeuvre).
— Pas d'Utilisation Commerciale (**NC**) : vous n'avez pas le droit d'utiliser cette création à des fins commerciales.
— Partage des Conditions Initiales à l'Identique (**SA**) : si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Table des matières

I Generalities concerning ELU610 & introduction 5

C1 : Introduction to languages & logic 7

II Regular expressions, automata, formal grammars 19

C2-4, TP1-3 : Formal languages - TODO 21

III λ-calculus, functional programming, compilation, typing, OCaml 71

C5 : The λ-calculus : Mathematical modeling of functions . . 73

TD1 : Let's practice rewriting 83

TD2 : Let's practice induction 85

C6 : Functional programming : Introduction to OCaml 87

TP4-6 : Discovering OCaml 97

C7 : Compilation : A crash course 103

Lecture notes – Compilation with OCaml 113

TD3, TP7-8 : Let's practice compilation 119

IV Logics 126

C8-11 : First order logic, description logic, Hoare logic 127

TD4-5 : Propositional logic, First order logic 157

TP9-10 : Prolog language 159

TD6 : Programming languages semantics 165

Première partie

Generalities concerning ELU610 & introduction

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Introduction to languages & logic

ELU 610 – C1
1st semester 2019

An introduction to...

- mathematical tools for computer science
- two new programming paradigms
- compilation and typing
- tools for knowledge representation

Organization

0. Introduction (C1, now)
1. Regular expressions, automata, formal grammars
   - C2-4, TP1-3
   - Éric Cousin – office D03-014, `eric.cousin@imt-atlantique.fr`
2. λ-calculus, functional programming, compilation, typing, OCaml
   - C5-7, TD1-2, TP4-9
   - Fabien Dagnat – office D03-120
   - Jean-Christophe Bach – office D03-124
     `{fabien.dagnat,jc.bach}@imt-atlantique.fr`
3. Logics
   - C8-11, TD3-5, TP10-11
   - Yannis Haralambous – office D03-118,
     `yannis.haralambous@imt-atlantique.fr`

Evaluation

- Theory
  - each part is evaluated at the end of ELU610, june 6th
- Practice
  - a stack language compiler written in OCaml
  - 2 members per compiler group
  - not handing (or contributing to) the compiler is eliminatory

Introduction to *Languages & logic*

Why *Languages & logic*?

- what relationship between language and logic?
- why that content in a lecture?
- why studying formal systems and abstractions instead of practical activities?

Today's lecture: motivations for L&L

+ few formal definitions
+ some terminology

Abstractions *vs* practical activities

- Playing with (changes of) abstractions is part of engineer's core activity
- Reasonable pedagogical choice

Motivations

- Complex software systems
- Critical systems
- Need of trusted software for trusted systems and services

⇒ designing, developing and verifying software

{quality, safety, security} by design

⇒ need of tools and methodologies

How to solve a problem?

A problem well stated is on its way to solution

Bergson, XXth

- What does *stated* means?
- What is a *well* stated problem?
- Then *solving* it. . .

Reflection on reflection

- What tools do we have to state problems?
  - natural language, pictures/drawings, mathematics, programs
  - media (audio – voice, paper – writings, drawings, electronics. . .)
- How do we state problems?
  - identification
  - selection
  - description (with a sound, a word, a picture, a formula, a program, etc.)

Identifying

- Recognize, invent
  - with respect to previous identical experiences
  - with respect to previous close experiences
- Similarities, metaphors, links, comparisons
- Exact, approximate, complement (lack of), . . .
- An invention from scratch is rare, . . . (is it even possible?)

Naming

In the beginning was the Word

John the Apostle, I^st (?)

Mal nommer les choses, c'est ajouter au malheur du monde

Albert Camus

- The importance of choosing right names
  - ambiguities, vagueness
  - method overriding/overloading

Selecting

- Among the identified *things*, which one to keep?
  - all?
  - the useful one? Useful relatively to an *intent* (the problem to solve)
- Ockham's razor

  Entities must not be multiplied beyond necessity.
  Plurality should not be posited without necessity.

William of Ockham, XIVth

An usual interpretation is: "when you have two competing theories that make exactly the same predictions, the simpler one is the better"

To describe an idea in order to:

- Transmit (in time, to others, to oneself)
- Handle, work with
- To interpret, with risks like:
  - incomprehension. . . easy; "I'm able to detect when I do not understand!"
    - lost languages or writings
  - misunderstanding. . . "I understood something, but not the intent of the transmitter" hard to detect. . . but a factor of innovation

Our ability to identify, to select and to name depends on our toolbox of descriptions

(Scientific) progress is a consequence of this virtuous principle:

> *We are like dwarfs on the shoulders of giants, so that we can see more than they, and things at a greater distance, not by virtue of any sharpness of sight on our part, or any physical distinction, but because we are carried high and raised up by their giant size*
>
> *Bernard de Chartres, XIIth*

- Sounds
  - problems: trace, memory, transmission, sophisms (validity, correctness)
- Writings
  - problems: sophisms (validity, correctness)
- Graphical
  - problems: validity (interpretation/semantics)
- Mathematics
  - problems: accessibility, calculability/completeness
- Computers
  - problems: validity – 4-colors theorem (?), size of problems, calculability/completeness

- Modeling
  - abstracting a problem, stating it. . .
  - simplifying, hiding details
- What for?
  - solving problems (of course!)
  - helping to think
  - mastering complexity
  - validating
  - verifying
- How to. . .
  - . . . express a model / represent concepts?

  ⇒ with languages
  - . . . how to "solve" a problem with models? (how to reason?)

  ⇒ with logics

Modeling languages

- Mathematics
  - rich, precise, rigorous
  - possess powerful transformation tools
  - ex.: from $5 + x = 8$ one reduce $x = 8 - 5$, hence $x = 3$!
- Maps, pictures
  - rich, abstract
  - "One picture is worth ten thousand words"
  - transformations (3D algorithms, drawing constraints)
- Simulations
  - models (french *maquettes*), prototypes
  - actors, virtual reality, ...
- ...

Languages as mechanisms that help thinking

- Modeling
  - choose a good language (to be able to express concepts)
  - symbols, graphical notations
  - mechanisms, operations
- Example in math, using "algebra" (no verb!?)
  A square has a surface a. What is the length of its side?
  - x is the length of the side
  - x is such that $x^2 = a$
  - ...do not forget $x \geqslant 0$!
- Defining a language needs time

Example : math notations in history [Nic94]

Authors	$+$	$=$	x	$2x^2 = 3x + 5$
Chuquet (XVth)	$\bar{p}$		$1, 2, 3$	2^2 egaulx a 3^1 $\bar{p}$ 5
Stifel (XVIth)	$+$		x, z, a	$2z$ acquatus $3x + 5$
Cardan (XVIth)	$\bar{p}$		*co*, *ce*, *cu*	2 *ce* equale a 3 *co* $\bar{p}$ 5
Bombelli (XVIth)	$\bar{p}$		$\underline{1}, \underline{2}, \underline{3}$	$\frac{2}{2}$ equale a $\frac{1}{3}$ $\bar{p}$ 5
Stevin (XVIth)	$+$		①, ②, ③	2② aequatus 3① $+ 5$
Viète (late XVIth)	$+$		A, Aq, Ac	2 in *Aq* aequatur 3 in *A* $+$ 5 plano
Neper (XVIIth)	$+$	$==$	R, Q, C	$2Q == 3R + 5$
Harriot (1631)	$+$	$==$	a, aa, aaa	$2aa == 3a + 5p$
Hérigone (1634)	$+$	$2/2$	$a, a2, a3$	$2a2$ $2/2$ $3a + 5p$
Descartes (1637)	$+$	∞	z, zz, z^3	$2zz \infty 3z + 5$

Good modeling language? Criteria

- No ambiguities (2/2, 1, 2, etc.)
- Generalizable (1 to n unknowns)
- Simple (5 plano is redundant)
- Economical (short)
- Ease communication/easy to learn

Cognitive gap: naming what is known is natural; naming the unknown, less...!

Language

Everything is about language

- to express
- to reason about

Which language to use?

- universal language? ⇒ universal tool
- specialized languages? ⇒ dedicated tools
- natural languages? ⇒ tools?

Natural language

Why not using natural (not formal) language?

- ambiguities
- under-specification (understatement, implicit)
- over-specification (redundancy)
- noise
- easy to have contradiction
- difficult to have the right level of specification

⇒ difficult to reason with natural languages

Specialized languages

DSL: Domain Specific Language

- special purpose languages...
- on purpose language limitations (Controlled Natural Language)

⇒ Specialized tools for reasoning, transforming, proving, ...

Formal language

- removing/avoiding ambiguities
- automating reasoning (partially)

⇒ useful for software verification

- formality with 3 levels of correctness:
  1. $2x+ = 8-$ (syntactic)
  2. $2x = 10 \Rightarrow x = 10 - 2$ (transformation)
  3. $2x = 10 \Rightarrow x = 10/2 = 5$ (intention)

Levels 1 and 2 can be automated.
Level 3 requires interpretation, and some kind of agreement (consensus); is the problem well stated?

> *[A proof] is a social process that determines whether mathematicians feel confident about a theorem [DLP78]*

Defining a language

- Language = syntax + semantics
- Syntax
  - we have tools to describe syntax without any interpretation:
  - ⇒ formal grammars
  - writing programs which recognize syntactically correct programs
  - ⇒ compilers
- Semantics
  - what happens when executing
  - two languages can have the same syntax but different semantics
  - interpretation
  - set of rules, transformations and constraints attached to syntax

Note: reasoning and deduction are a purely syntactical process (⇒ useful for automation. . .)

Defining a language (*teaser* of next lecture)

- Formal definitions
  - language, alphabet, symbols, terms, . . .
  - focus of CS on *finitely generated* languages
- *formal language theory* (study and classification of languages) [ALSU06, HMU06]
  - focus on how to define languages and (efficiently) recognize terms

    see http://en.wikipedia.org/wiki/Formal_language
- . . . and many other interesting language-related things

⇒ Do not miss Éric Cousin's lecture! It is mandatory to understand how we work with languages and compilation in CS

Defining a syntax (another *teaser* of next lecture)

- A syntax? *Two* syntaxes: a *concrete* one and an *abstract* one
- Concrete syntax
  - defined by a grammar, using BNF/EBNF

    see http://en.wikipedia.org/wiki/Backus-Naur_Form
  - focus on interaction with the user
    - must be readable, efficient, . . .
    - must solve the ambiguities, priorities, associativity, . . .

    see http://www.infoq.com/presentations/Language-Design
- Abstract syntax
  - the *essential* content of a sentence
  - aimed at being used by any tool manipulating terms
  - defined by a signature (using eventually a BNF/EBNF grammar)
- Understanding syntaxes is necessary to build a compiler

⇒ Do not miss Éric Cousin's next lecture

Syntax and semantics

We talked a lot about syntax, few about semantics.
Where is the semantics?

- in your mind first (we are interpreters)
- in the set of rules, transformations, constraints that we attach to a syntax (ex. $+$ is associative and commutative)
- in mappings we make to a well known world with its own syntax and semantics (ex. mathematics)

Giving meaning

- There is mainly three ways of defining the semantics of a term
  1. Axiomatic semantics: some logical assertions states properties of terms
  2. Denotational semantics: each term is mapped to an object of a known space
  3. Operational semantics: how computation behaves (the sequence of states)
- Not the only ones

see http://en.wikipedia.org/wiki/Semantics_(computer_science)

Axiomatic semantics

- Defined by systems of equations describing the effect of each syntactic construction to logical assertions
- Gives a macroscopic vision of the meaning (generally partial)
- Used to study properties of: consistency, completeness, compositionality, ...
- The most well-known, Hoare triple
  - $\{Pre\}T\{Post\}$ means if *Pre* is true before the execution of *T* and *T* terminates then *Post* is true after its execution

see http://en.wikipedia.org/wiki/Hoare_logic

Denotational semantics

- Defined by a projection in a (known) mathematical space (sets, universal algebra, domain, category, ...)
- Gives an abstract vision of the meaning
- Used to study meta-theory: equivalence of terms, fixed-point theory, ...
- Often given by a projection called an interpretation and denoted $[\![.]\!]$ or $\mathcal{I}(.)$
- Often requires compositionality, the meaning of a term is the composition of the meaning of its subterms

see http://en.wikipedia.org/wiki/Denotational_semantics

Operational semantics

- Each term either reduce to another (smaller) term or is a value
- Defined by computation rules (rewriting)
- Can be small-step or big-step
- Gives a microscopic vision of the meaning
- Used to study properties of: termination, non-determinism, ...
- The one we will use
- More precisely, we will use transitions systems (*a.k.a.* reduction)

see http://en.wikipedia.org/wiki/Operational_semantics

⚠ The following slides might be a bit shuffled

Do you know those words? Do you know they meaning?

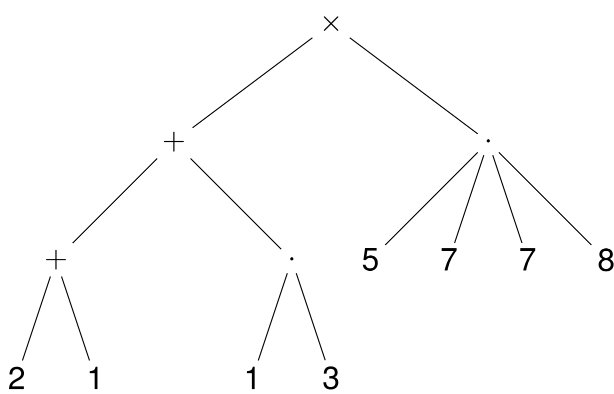
- verification, validation
- term, metaterm
- variable, metavariable
- transition system

An interpretation...

Verification
: checking that the rules of the formal systems are properly used. Internal to a model, a description system and its use.

Validation
: comparing two (2) models to check that the one to be validated *gives the same answer* that the one of reference.

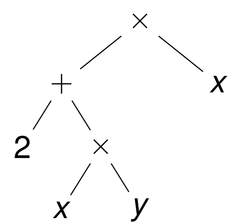
- Terms are trees where each constructor is a node and each of its direct sub-terms is a child



- We use names to
  - represent a set of terms: metavariable; $c(M_1, M_2)$, $E_1 \times (3 + E_2)$
  - represent an unknown part of a term: variable; $c(x)$, $x \times y$
- Metavariables are not part of the syntax and used only inside metaterms
  - a metaterm is a set of term
  - useful to manipulate or to describe properties of sets of terms
- Variables are part of the term
  - syntax must be extended (see next slide)
  - a variable may occur several times within a term
  - the meaning of a variable is given by replacing all its occurrences by a term
  - a term T containing x can be viewed as a function from term to term

Variables

- Terms may also contain variables from a denumerable set $\mathcal{X}$
  - we suppose $\Sigma \cap \mathcal{X} = \varnothing$ and the arity of variables is 0
- $T_{\Sigma \cup \mathcal{X}}$ is denoted $T_{\Sigma}[\mathcal{X}]$
- A term without variable is a ground term (or closed term)
- Variables are leafs (as nullary constructors)



- The meaning of a variable is given by substitutions

Transition system

A transitions system is a pair $(S, \rightarrow)$ of a set S (of states) and a binary relation $\rightarrow$ of S ($\rightarrow \subset S \times S$).

A pair (p, q) of $\rightarrow$ is noted infix $p \rightarrow q$ and we speak of a transition from state p to state q.

programs $\leftrightarrow$ transition systems

More terminology

This introduction could have much more vocabulary. Some will (should) be in the next lectures

- normal term, normal form
- trace, reduction sequence
- (non-)determinism
- strong normalization
- weakly normalizing
- confluence
- reflexive transitive closure
- ...

But we are humans... If you hear *strange* (unknown) words which are not defined during the lectures, please tell us.

Conclusion

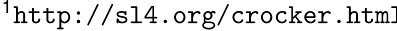
- an introduction to motivate ELU610
- now, you should understand why there is a lecture combining language and logic
- few intuitions before more formal lectures and definitions
- don't be scary: there are also practical and concrete parts (programming!) to show you that it is useful

Éric Cousin Regular expressions, automata, formal grammars

JC Bach λ-calculus, introduction to functional programming, compilation and typing (OCaml language)

. . . with Fabien Dagnat

Yannis Haralambous Variation about logics



Important note: if you do not understand something or if you disagree with us, please say it and ask your questions. We won't bite you, and we follow Crocker's rules[1].

We cannot answer the questions you do not ask. . .

[1] `http://sl4.org/crocker.html`

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman.
Compilers: Principles, Techniques, and Tools (2nd Edition).
Addison Wesley, August 2006.

Richard A DeMillo, Richard J Lipton, and Alan J Perlis.
Social Processes and Proofs of Theorems and Programs (Revised Version).
1978.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman.
Introduction to Automata Theory, Languages, and Computation (3rd Edition).
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

Jean-Marie Nicolle.
Histoire des méthodes scientifiques.
Bréal, 1994.

Deuxième partie

Regular expressions, automata, formal grammars

Eric Cousin
eric.cousin@imt-atlantique.fr

ELU610 - Langages formels

Support de cours

Version 2018.2 - 24/09/2018
Formation d'ingénieur
Année scolaire 2018-2019

Contenu

Ce livret rassemble différents documents relatifs aux enseignements menés dans la première partie de l'UE ELU610 et consacrés à l'étude des langages formels. Il contient, dans l'ordre :

1. Le support du cours (copie des diapos). A noter qu'u**ne version commentée est disponible sur Moodle.**
2. L'énoncé du TP sur les expressions régulières.
3. L'énoncé du TP sur les automates d'états finis.
4. Deux énoncés d'exercices concernant à la fois les expressions régulières, les automates d'états finis et les grammaires formelles, permettant de s'exercer via un travail autonome.
5. L'énoncé du TP sur les grammaires formelles.
6. Un petit récapitulatif des objectifs pédagogiques de cette partie d'ELU610 permettant de faire le point sur son apprentissage.

Tous ces documents, ainsi que des compléments, sont disponibles sur l'espace Moodle du module ELU610.

RAPPELS PÉDAGOGIQUES

« Cette UE est une introduction à certains outils mathématiques de base pour l'informatique (les **langages formels**, la logique du premier ordre, le λ-calcul typé), à deux nouveaux types de programmation : la fonctionnelle et la programmation logique et à la **compilation** »

PLAN DU COURS

LES LANGAGES FORMELS, OUTIL DE L'INGÉNIEUR
LANGAGES FORMELS
UN PEU DE THÉORIE
EXPRESSIONS RÉGULIÈRES
AUTOMATES D'ÉTATS FINIS
LANGAGES RÉGULIERS
TRAITEMENTS SÉMANTIQUES
GRAMMAIRES FORMELLES
TRAITEMENTS SÉMANTIQUES (2)
PRINCIPALES PHASES DE LA COMPILATION
APPLICATION : LA MINI-CALCULETTE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

LES LANGAGES FORMELS, OUTIL DE L'INGÉNIEUR

EX : UTILISER UN LANGAGE DE PROGRAMMATION 5

1. Introduction

This reference manual describes the Python programming language. It is not intended as a tutorial.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here. If you would like to see a more formal definition of the language, maybe you could volunteer your time — or invent a cloning machine :-).

1.2. Notation

The descriptions of lexical analysis and syntax use a modified BNF grammar notation. This uses the following style of definition:

```
name      ::=  lc_letter (lc_letter | "_")*
lc_letter ::=  "a"..."z"
```

The first line says that a `name` is an `lc_letter` followed by a sequence of zero or more `lc_letter`s and underscores. An `lc_letter` in turn is any of the single characters `'a'` through `'z'`. (This rule is actually adhered to for the names defined in lexical and grammar rules in this document.)

Each rule begins with a name (which is the name defined by the rule) and `::=`. A vertical bar (`|`) is used to separate alternatives; it is the least binding operator in this notation. A star (`*`) means zero or more repetitions of the preceding item; likewise, a plus (`+`) means one or more repetitions, and a phrase enclosed in square brackets (`[ ]`) means zero or one occurrences (in other words, the enclosed phrase is optional). The `*` and `+` operators bind as tightly as possible; parentheses are used for grouping. Literal strings are enclosed in quotes. White space is only meaningful to separate tokens. Rules are normally contained on a single line; rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

In lexical definitions (as the example above), two more conventions are used: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters. A phrase between angular brackets (`<...>`) gives an informal description of the symbol defined; e.g., this could be used to describe the notion of 'control character' if needed.

Even though the notation used is almost the same, there is a big difference between the meaning of lexical and syntactic definitions: a lexical definition operates on the individual characters of the input source, while a syntax definition operates on the stream of tokens generated by the lexical analysis. All uses of BNF in the next chapter ("Lexical Analysis") are lexical definitions; uses in subsequent chapters are syntactic definitions.

(Cf http://docs.python.org/2/reference/)

EX : UTILISER UN LANGAGE DE PROGRAMMATION (SUITE) 6

2.3. Identifiers and keywords

Identifiers (also referred to as *names*) are described by the following lexical definitions:

```
identifier ::=  (letter|"_") (letter | digit | "_")*
letter     ::=  lowercase | uppercase
lowercase  ::=  "a"..."z"
uppercase  ::=  "A"..."Z"
digit      ::=  "0"..."9"
```

Identifiers are unlimited in length. Case is significant.

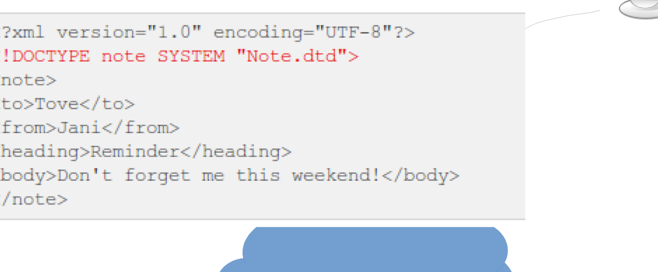
2.3.1. Keywords

6.12. The `import` statement

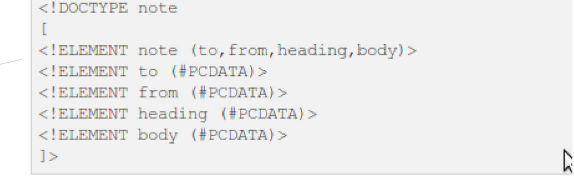
```
import_stmt     ::=  "import" module ["as" name] ( "," module ["as" name] )*
                     | "from" relative_module "import" identifier ["as" name]
                     ( "," identifier ["as" name] )*
                     | "from" relative_module "import" "(" identifier ["as" name]
                     ( "," identifier ["as" name] )* [","] ")"
                     | "from" module "import" "*"
module          ::=  (identifier ".")* identifier
relative_module ::=  "."* module | "."+
name            ::=  identifier
```

Import statements are executed in two steps: (1) find a module, and initialize it if

EX : TRANSFÉRER DES DONNÉES 7



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```



```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```


EX : TRAITER DES CHAÎNES DE CARACTÈRES 8

Typique : vérifier que l'utilisateur a bien fourni une adresse de messagerie, une date, un numéro de portable…

```
if (mail.matches("[a-z][-_.a-z]*@[a-z0-9.-]{2,}\\.[a-z]{2,4}")) ...
if (date.matches("[0-3][0-9]/[01][0-9]/[0-9]{4}")) …
if (date.matches("(0|\\+33 )[67]((-| )?[0-9]{2}){4}")) …
```

en une seule instruction !

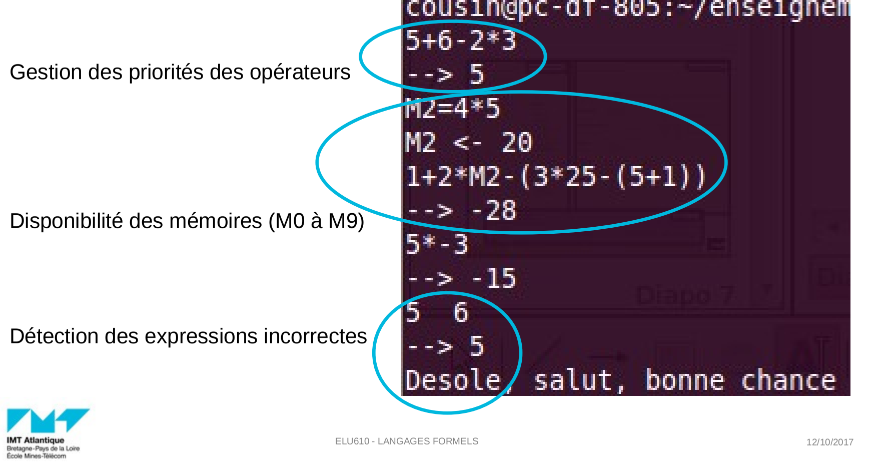
EX : TRAITER UN LANGAGE

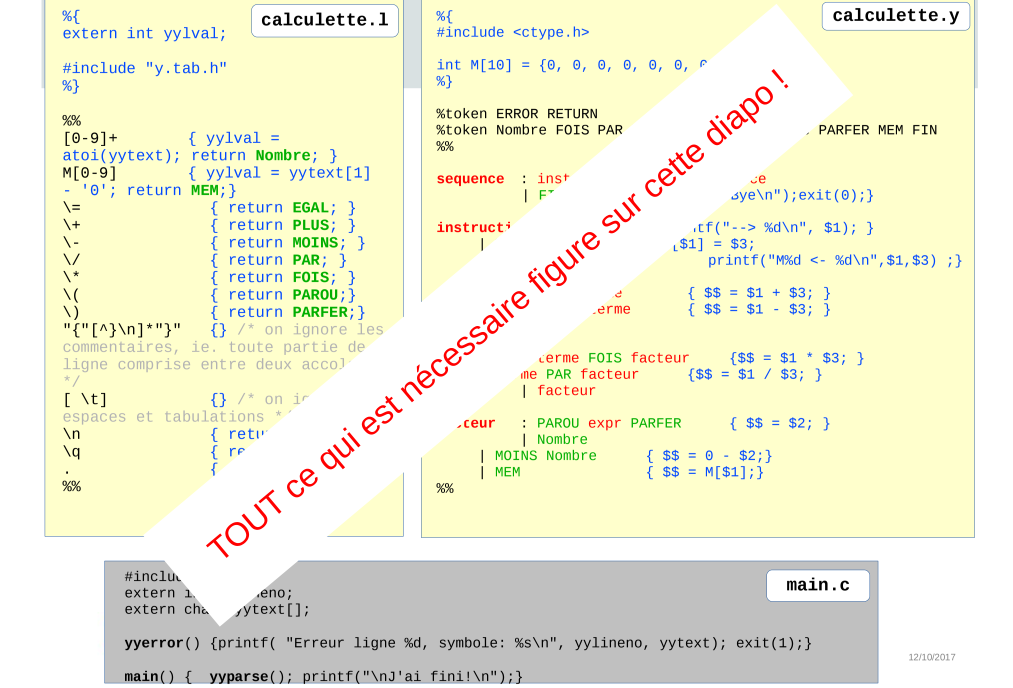
Effort pour réaliser une mini calculette interactive ?

Gestion des priorités des opérateurs

Disponibilité des mémoires (M0 à M9)

Détection des expressions incorrectes





UTILITÉ DES LANGAGES FORMELS POUR L'INGÉNIEUR : RÉCAPITULATIF

- Comprendre et utiliser des spécifications de langages
- Mieux comprendre les messages d'erreur du compilateur
- Spécifier/décrire un langage
- Élaborer un traitement de langage
- Concevoir et mettre en œuvre un système réactif
- …

LANGAGES FORMELS (ELU610)

Objectifs et contenus

OBJECTIFS D'APPRENTISSAGE

A l'issue de cette partie de ELU610, vous devriez être capable de :

- Comprendre et utiliser des spécifications de langage
- Formaliser un langage simple avec Expressions régulières (E.R.), Automates d'états finis (A.E.F), Grammaires formelles (G.F)
- Modéliser le comportement d'un système réactif par un A.E.F
- Utiliser des outils manipulant des E.R
- Expliquer – dans le processus de compilation - les rôles respectifs des analyses lexicale, syntaxique et sémantique

A court terme, vous devriez être en mesure de mobiliser des outils manipulant des G.F

NB : voir document d'auto-évaluation du livret

CONTENU

C1 Langages formels, langages réguliers, expressions régulières

TP1 – expression régulières

C2 Automates d'états finis

TP2 – automates d'états finis

C3 Grammaires hors-contexte

TP3 – grammaires formelles

UN PEU DE THÉORIE & FORMALISME

Un (plusieurs) méta-modèle(s) pour travailler sur les langages.

Comment spécifier un langage ?

Comment reconnaître le langage ?

Comment formaliser le traitement d'un langage

Langages formels, Automates, Expressions régulières, Grammaires formelles...

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

LANGAGE & FORMEL, DÉFINITIONS USUELLES

fr.wiktionary.org, consulté le 17/02/15

Langage :

« (Linguistique) Faculté de mettre en œuvre un système de signes linguistiques, qui constituent la langue, permettant la communication et l'expression de la pensée »

Communication avec une personne... ou une machine

Formel :

« Qui est exprimé avec une clarté et une précision parfaites », afin d'exclure toute méprise ou équivoque.

JAVA LANGAGE SPEC. 17

3.7 Comments

There are two kinds of comments:

`/* text */` A traditional comment: all the text from the ASCII characters /* to the ASCII characters */ is ignored (as in C and C++).
`// text` A end-of-line comment: all the text from the ASCII characters // to the end of the line is ignored (as in C++).

These comments are formally specified by the following productions:

```
Comment:
       TraditionalComment
       EndOfLineComment

   TraditionalComment:
       /* NotStar CommentTail

   EndOfLineComment:
       // CharactersInLineopt LineTerminator

   CommentTail:
       * CommentTailStar
       NotStar CommentTail

   CommentTailStar:
       /
       * CommentTailStar
       NotStarNotSlash CommentTail

   NotStar:
       InputCharacter but not *
       LineTerminator

   NotStarNotSlash:
       InputCharacter but not * or /
       LineTerminator

   CharactersInLine:
       InputCharacter
       CharactersInLine InputCharacter
```

Spécification Formelle

Langue naturelle (informel)

These productions imply all of the following properties:

- Comments do not nest.
- /* and */ have no special meaning in comments that begin with //.
- // has no special meaning in comments that begin with /* or /**.

As a result, the text:
/* this comment /* // /** ends here: */
is a single complete comment.

The lexical grammar implies that comments do not occur within character literals (§3.10.4) or string literals (§3.10.5).

« FORMEL »… VOUS AVEZ DIT « FORMEL » 18

Autre définition (tjs sur wiktionary) : « Qui s'attache à la forme »

136

CAC40

LANGAGE : DÉFINITION FORMELLE 19

Alphabet : ensemble fini de symboles (lettres) -> X

Mot : suite finie et ordonnée de lettres de X

X^* = ensemble de tous les mots sur X

Langage = ensemble de mots définis sur le même alphabet (-> partie de X^*)

Opérations sur les mots

Longueur

Concaténation (opération interne, associative, non commutative, élément neutre = mot vide = ε)

PREMIER MÉTAMODÈLE : LES ENSEMBLES 20

L1 = {a, b, aa, aab} *définition en extension*

L2 = $\{x \in X^* \,/\, |x| \text{ est pair}\}$ *définition par compréhension (intension)*

L3 = $\{x \in \{a,b\}^* \,/\, |x|_a = |x|_b\}$

L4 = {ε}

L5 = ∅ *langage vide*

L6 = $\{w \in X^* \,/\, w \notin L2\}$ *complément d'un langage*

L7 = L1 ∪ L2 *union de langages*

L8 = L1 . L2 = L1 L2
= $\{w \in X^* \,/\, w = a_1\, a_2,\ a_1 \in L1 \text{ et } a_2 \in L2\}$ *concaténation de langages*

Ce métamodèle est assez lourd à utiliser

EXPRESSIONS RÉGULIÈRES (OU EXPRESSIONS RATIONNELLES)

Un méta-modèle spécialement créé pour définir des langages « simples »

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

EXEMPLE D'UTILISATION : URL

Exemples d'URL:

www.telecom-bretagne.eu

http://machin.fr

Comment vérifier qu'une chaîne de caractères entrée par un utilisateur ressemble bien à une URL de site Web ?

Protocole : http, https, ou implicite

Sous-domaines et domaine 2eme niveau : des lettres, des chiffres, certaines ponctuations,....

Domaine premier niveau : 2 ou 3 lettres

Les différents composants sont séparés par des points

```
"(http://|https://)?[a-zA-Z0-9][a-zA-Z0-9_.-]*[a-zA-Z0-9]\.[a-z]{2,4}"
```


EXPR. RÉGULIÈRES : DÉFINITION DE BASE

Les expressions régulières (E.R.) sur un alphabet X et les ensembles (langages) qu'elles représentent sont définies récursivement comme suit :

Ø est une E.R. et représente le langage vide

ε est une E.R. et représente le langage {ε}

Si a est un symbole de X, **a** est une E.R. et représente le langage {a}

Si R_1 et R_2 sont deux E.R. représentant les langages L_1 et L_2, alors :

$R_1|R_2$ est une E.R. représentant la $L_1 \cup L_2$

R_1R_2 est une E.R représentant $L_1.L_2$

R_1^* est une E.R. représentant L_1^*

EXEMPLES

Chiffres (numération en base 10) : C=0|1|2|...|9

Tous les mots définis sur X={a1, a2, ...an} :
X*=(a1|a2|...|an)*

(a|b*)c={ac,c,bc,bbc,bbbc,...,b....bc,...}

CONVENTIONS D'ÉCRITURE

Permettent de faciliter l'écriture,

… mais ne changent rien à la définition précédente

Ex : une ou plusieurs occurrences

a^+={a,aa,aaa,...} alors que a^*={ε,a,aa,aaa,...}

$R^+ = RR^* = R^*R$

Ex : k occurrences

$R^4 = RRRR$

$(a|bc)^2$=(a|bc)(a|bc)=aa|abc|bca|bcbc
soit {aa,abc,bca,bcbc}

EXEMPLES

a+ = aa* = a*a

Nombre réel : R = C+'.'C*

À comparer avec

R = C*'.'C*

R = C+'.'C+

AUTRES NOTATIONS (CF PERL)

D'autres extensions des notations sont acceptées par les outils qui manipulent les expressions régulières. Par exemple :

nb d'occurrences entre 1 et 4 (inclus) : a{1,4}

optionnel : a?

au moins 4 occurrences : a{4,}

ensembles de caractères : [a-zA-Z]

classes de caractères : \d (un chiffre = [0-9]), \D (tout sauf un chiffre), \s (espace, tabulation ou newline), ...

On retrouve plusieurs types de notation, mais les concepts sont strictement identiques.

Par exemple, pour certains outils [a] signifie (a|ε)

UTILISATIONS PRATIQUES

Vérifier la conformité d'une saisie de l'utilisateur

Ex : [0-3]?[0-9]/[a-z][a-z][a-z]/$[0-9]^4$

Définir des séquences d'événements attendues ou erronées

Chercher/détecter des séquences précises (→ filtrage)

Recherche d 'un motif dans un fichier texte

Valider des données (fusion de BD, data-mining, ...)

...

SPÉCIF DE CHAÎNES DE CARACTÈRES 29

Comment décrire les chaînes de caractères correspondant à :

Un nombre réel qui n'ait pas de zéro non significatif (à gauche et à droite) ?

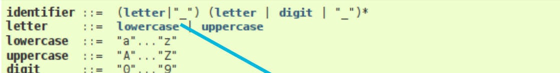
Une date ?

Une adresse de courrier électronique ?

Une adresse Web ?

Un numéro de téléphone ?

EX POUR PYTHON 30



2.3. Identifiers and keywords

Identifiers (also referred to as *names*) are described by the following lexical definitions:

```
identifier ::=  (letter|"_") (letter | digit | "_")*
letter     ::=  lowercase | uppercase
lowercase  ::=  "a"..."z"
uppercase  ::=  "A"..."Z"
digit      ::=  "0"..."9"
```

Identifiers are unlimited in length. Case is significant.

2.3.1. Keywords

SÉQUENCE D'ÉVÉNEMENTS : L'AUTOCOMMUTATEUR 31

Alphabet

D : Décrocher le combiné

R : Raccrocher le combiné

C : Composer un chiffre

Modélisation

Comportement d'un abonné

(DC*R)*

rque : ni (DC+R)* ni (DC*R)+

Établissement d'une communication entre deux abonnés

D_1C_1*(R_1|D_2(R_1R_2|R_2R_1))

UTILISATION SIMPLE EN JAVA 32

Java™ Platform Standard Ed. 6

All Classes

Packages
java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event

StrictMath
String
StringBuffer
StringBufferInputStream
StringBuilder
StringCharacterIterator
StringContent
StringHolder
StringIndexOutOfBoundsException
StringMonitor
StringMonitorMBean
StringNameHelper

a string derived from this string by replacing every occurrence of oldChar with newChar.

matches

`public boolean matches(String regex)`

Tells whether or not this string matches the given regular expression.

An invocation of this method of the form *str*.matches(*regex*) yields exactly the same result as the expression

Pattern.matches(*regex*, *str*)

Parameters:
regex - the regular expression to which this string is to be matched
Returns:
true if, and only if, this string matches the given regular expression
Throws:
PatternSyntaxException - if the regular expression's syntax is invalid
Since:
1.4
See Also:
Pattern

contains

```
if (chaine.matches("[a-zA-Z.]+@[a-z]+\\.[a-z]+")) ...
```

LA CLASSE `java.util.regexp` 33

```java
import java.util.regex.*;
public class Ex1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
      try {
          Pattern p = Pattern.compile ("a*b|c");
          tring entree = "aabbbcab";
          Matcher m = p.matcher(entree);
          while (m.find())
            System.out.println(entree.substring(m.start(), m.end()));
           }
      catch (PatternSyntaxException pse){}
    }
}
```

→

```
aab
b
b
c
ab
```

ELU610 - LANGAGES FORMELS 12/10/2017

LE MÊME, EN PYTHON 34

```python
import re
r = re.compile(r"a*b|c")
for m in r.finditer("aabbbcab") :
     print m.group()
```

→

```
aab
b
b
c
ab
```

ELU610 - LANGAGES FORMELS 12/10/2017

LA COMMANDE `grep` 35

Permet d'afficher les lignes d'un fichier contenant une chaîne de caractères correspondant à une expression régulière

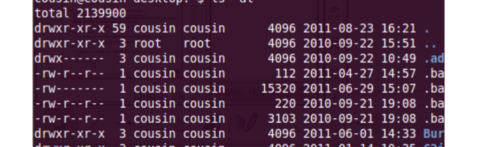
Exemples

appels à `throw` dans mes classes java :
`grep throw *.java`

définition de constantes dont le nom se termine par VAL dans mes fichiers C :
`grep '#define .*VAL' *.c`

fichiers dont root est le propriétaire :
`ls -al | grep root`

(ou dont le nom contient `root`)



ELU610 - LANGAGES FORMELS 12/10/2017

LA COMMANDE `awk`

```
cousin@cousin-desktop:~$ ls -al
total 2139900
drwxr-xr-x 59 cousin cousin      4096 2011-08-23 16:21 .
drwxr-xr-x  3 root   root        4096 2010-09-22 15:51 ..
drwx------  3 cousin cousin      4096 2010-09-22 10:49 .ad
-rw-r--r--  1 cousin cousin       112 2011-04-27 14:57 .ba
-rw-------  1 cousin cousin     15320 2011-06-29 15:07 .ba
-rw-r--r--  1 cousin cousin       220 2010-09-21 19:08 .ba
-rw-r--r--  1 cousin cousin      3103 2010-09-21 19:08 .ba
drwxr-xr-x  3 cousin cousin      4096 2011-06-01 14:33 Bur
drwxr-xr-x  3 cousin cousin      4096 2011-01-14 10:35 C2i
```

Plus fort que `grep` : `awk` permet

De filtrer un fichier texte avec plusieurs E.R

D'appliquer des traitements différenciés selon les cas

De manipuler individuellement les « mots » des lignes

Exemples :

traitement par défaut = afficher l'enregistrement :
`awk /throw/ *.java`

Tailles de fichiers :
`ls -l | awk '$5 > 4000'`
`ls -l | awk '$6 > 2009 {sum += $5} END {print sum}'`

ELU610 - LANGAGES FORMELS 12/10/2017

EXPRESSIONS RÉGULIÈRES « À LA PERL » NOTION DE GROUPES

Quand on souhaite non seulement trouver des appariements entre le motif et la chaîne mais aussi manipuler des sous-parties de la chaîne ainsi filtrée

N'est pas disponible sur tous les outils

Utilisation de parenthèses pour structurer le « motif ». Par ex, `a((bc)(d))e` définit 4 **groupes**, numérotés par ordre d'apparition des parenthèses ouvrantes :

groupe 0 : `a((bc)(d))e` -> `abcde`

groupe 1 : `((bc)(d))` -> `bcd`

groupe 2 : `(bc)` -> `bc`

groupe 3 : `(d)` -> `d`

Attention : modifie fondamentalement le formalisme

UTILISATION DES GROUPES

Dans le traitement de l'expression filtrée

Ex : filtrage d'une adresse électronique
`([a-z]+)@([a-z.-]+)` => $1=destinataire, $2=hébergeur

Dans l'expression régulière elle-même

Ex : mots contenant une répétition d'une succession de 2 lettres (ali**baba**, **maman**, **papa**…)
`\S*(\S\S)\1\S*`

Attention : en général `(...){2}` ≠ `(...)\1`

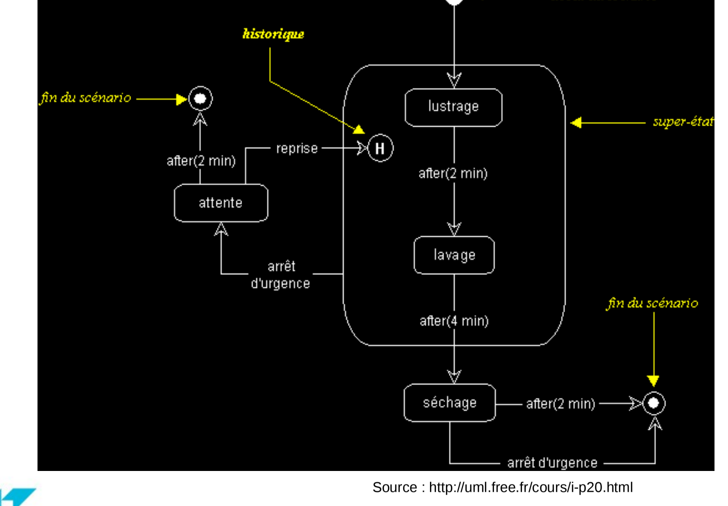
AUTOMATES D'ÉTATS FINIS

Une expression régulière permet de définir/spécifier un langage

Un automate d'états fini (AEF) offre en outre un moyen de tester de façon automatique si un mot appartient au langage reconnu

C'est aussi un méta-modèle comportemental

EX :DIAGRAMME ÉTATS-TRANSITIONS (UML)



Source : http://uml.free.fr/cours/i-p20.html

EX 2 : JOURNÉE D'UN ÉLÈVE REPRÉSENTÉE EN LTS
(COPYRIGHT INF447 / F. DAGNAT)

41

sereveiller
allerencours
sassoupir
ecouter
sendormir
sortircours
poser.question

Les états :

0 : Endormi
1 : Éveillé
2 : En cours

DÉFINITION

42

En théorie des langages,
un AEF est défini par <X,Q,D,A,d> où

X = alphabet

Q = ensemble fini d'états

D ⊂ Q ensemble des états de départ

A ⊂ Q, ensemble des états d'arrivée (ou terminaux)

d ensemble des transitions

NOTIONS FONDATRICES

43

Représentation graphique

a
b
a
b
a
a
ε

État, transition, état de départ, état terminal

NB : en LTS (cf ex2), la problématique état initial/final est différente

Représentation tabulaire
(cf table de transition)

	a	b	ε
→1	1, 2		
2	3		4
→3		2, 4	
4	4		

FONCTION PRINCIPALE DE L'AEF : RECONNAÎTRE LES MOTS DU LANGAGE

44

Chemin : succession de transitions adjacentes

Un mot est accepté par l'AEF ssi il existe un chemin allant d'un état de départ à un état terminal et correspondant au mot considéré.

On dit aussi que le mot est reconnu par l'AEF

Exemples

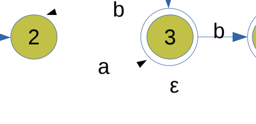
aaba : →1→2→3→2→3

aaab :

baaa :

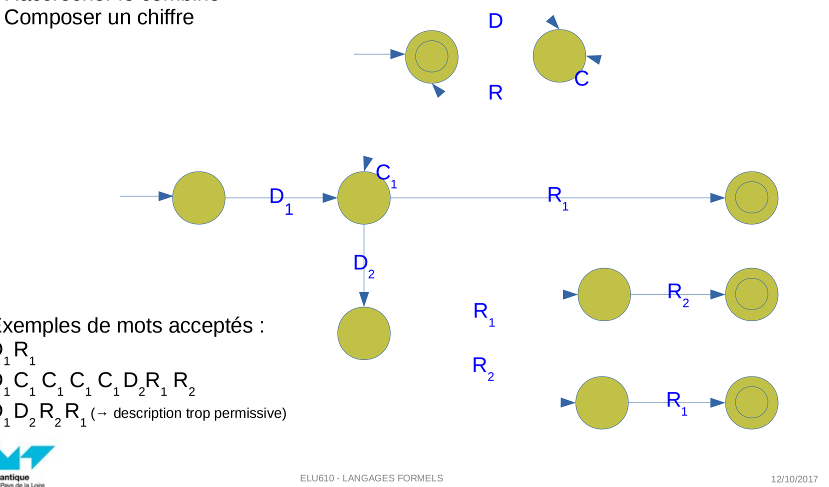
ε

Preuve



RETOUR SUR L'AUTOCOMMUTATEUR

Alphabet
D : Décrocher le combiné
R : Raccrocher le combiné
C : Composer un chiffre



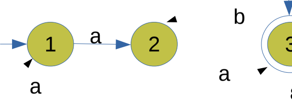
Exemples de mots acceptés :
D_1R_1
$D_1C_1C_1C_1C_1D_2R_1R_2$
$D_1D_2R_2R_1$ (→ description trop permissive)

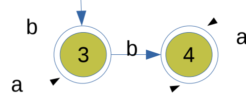
DÉTERMINISME

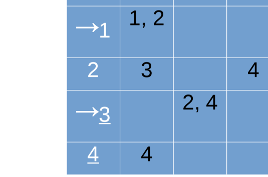
Une propriété fondamentale des AEF

Un AEF est dit déterministe ssi :
Il possède un seul état de départ
Il n'a pas de transition sur le mot vide
Pour tout état, il y a AU PLUS une transition sortante pour un symbole donné

Qu'en est-il de notre cas introductif ?







	a	b	ε
→1	1, 2		
2	3		4
→3		2, 4	
4	4		

Comment savoir si le mot aabb est accepté ?

PRINCIPE DE LA DÉTERMINISATION

Pour tout AEF non déterministe reconnaissant un langage L, il existe un AEF déterministe reconnaissant le même langage L.

Pour le trouver, il existe un algorithme. Celui-ci est en général disponible dans les outils qui travaillent avec les AEF. Cependant, comprendre cet algorithme permet de mieux comprendre le concept d'automate.

Algorithme permettant de construire la table de transition d'un AEFD A_2 correspondant à AEFND A_1 :

Chaque état A_2 représente un ensemble d'états de A_1

État de départ D = ens de tous les états de départ de A_1

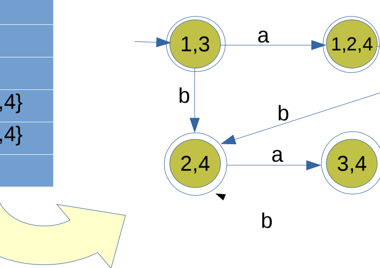
Pour chaque nouvel état $\{E_1, E_2,...E_n\}$ mis en évidence, on détermine pour chaque symbole de l'alphabet quels sont les états de A_1 accessibles depuis l'un des E_i, ce qui nous donne l'état suivant dans A_2

État d'arrivée = tout état correspondant à au moins un état final dans A_1

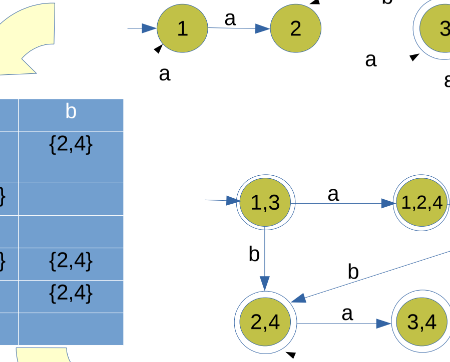
Attention : c'est un algorithme de complexité exponentielle

L'AEFD correspondant à un AEFND à n états peut avoir jusqu'à 2^n états

EXEMPLE DE DÉTERMINISATION



	a	b
→{1,3}	{1,2,4}	{2,4}
{1,2,4}	{1,2,3,4}	
{2,4}	{3,4}	
{1,2,3,4}	{1,2,3,4}	{2,4}
{3,4}	{4}	{2,4}
{4}	{4}	



Vérifions l'équivalence pour : aaba, aaab, baaa, ε, aabb

INTÉRÊT DE LA DÉTERMINISATION

Algorithme (et mise en œuvre) de la reconnaissance

plus simple

plus efficace : complexité algorithmique = O(n)

Surcoût de la déterminisation

peut être important,

mais payé une seule fois par automate (vs nb reconnaissances)

NB : c'est la même problématique que compiler vs interpréter

RETOUR SUR LES AUTOMATES *NON* DÉTERMINISTES

Attention : la reconnaissance d'un mot par un automate NON déterministe n'est pas aléatoire ou probabiliste. Soit il existe un chemin reconnaissant le mot analysé (et donc ce mot est accepté), soit il n'en existe pas (et il n'est pas accepté). En aucun la réponse oui/non ne dépend de l'ordre dans lequel l'automate essaie les différents chemins.

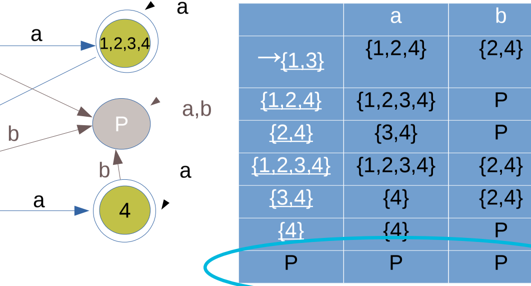
Lorsqu'il y a plusieurs transitions correspondant à la configuration courante (état, symbole lu), TOUTES ces possibilités doivent être envisagées pour pouvoir décréter qu'il n'existe pas de chemin reconnaissant le mot analysé.

Par contre, si plusieurs chemins acceptent un mot, on ne peut savoir lequel l'automate va choisir. Un AEFND est donc un « mauvais » outil dès lors que le chemin choisi aura de l'importance sur le traitement.

COMPLÉTUDE

Déf : un AEFD est dit complet ssi pour tout état, il y a EXACTEMENT une transition sortante pour CHAQUE symbole

Propr : algo pour rendre complet un AEFD



	a	b
→{1,3}	{1,2,4}	{2,4}
{1,2,4}	{1,2,3,4}	P
{2,4}	{3,4}	P
{1,2,3,4}	{1,2,3,4}	{2,4}
{3,4}	{4}	{2,4}
{4}	{4}	P
P	P	P

A ne pas oublier !

AEFDC : intérêt « opérationnel »

Ex : abaa

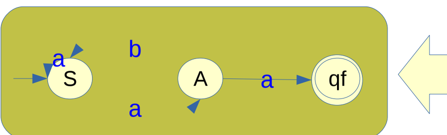
EXPRESSIONS RÉGULIÈRES ET AEF

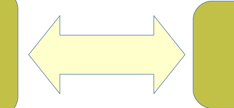
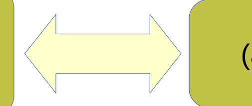
De l'E.R vers l'AEF

voir par ex http://www.regexper.com/

De l'AEF vers l'E.R

Remarque : le passage « à la main » d'un formalisme à l'autre peut être très difficile ou hors de portée humaine



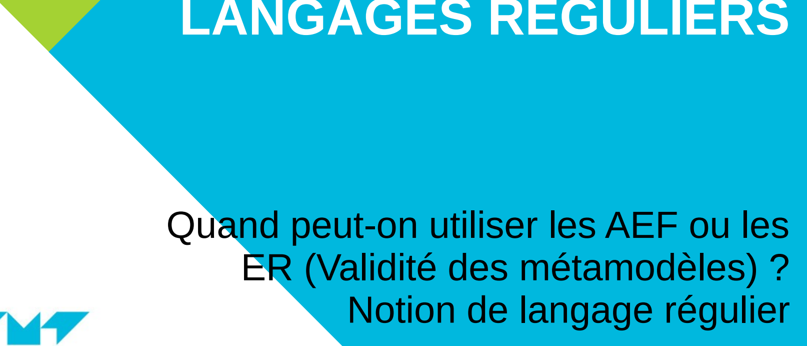



JAVA.UTIL.REGEX

54

```
import java.util.regex.*;
public class Ex1 {
    /**
     * @param args
     */
    public static void main(String[] args) {
        try {Pattern p = Pattern.compile ("a*b|c");
            String entree = "aabbbcab";
            Matcher m = p.matcher(entree);
            while (m.find())   System.out.println(entree.substring(m.start(), m.end()));
            }
        catch (PatternSyntaxException pse){}
    }
}
→
aab
b
b
c
ab
```

L'expression régulière est transformée en AEFD



LANGAGES RÉGULIERS

Quand peut-on utiliser les AEF ou les ER (Validité des métamodèles) ?
Notion de langage régulier

LANGAGES RÉGULIERS : CARACTÉRISATION

56

Un langage est régulier si on peut trouver une expression régulières ou un AEF le caractérisant.

Que conclure si on n'en trouve pas ?

Doit-on s'entêter à en trouver ?

Comment éviter de perdre son temps ?

Reconnaître que le langage « ressemble » à une combinaison de langages que l'on sait réguliers est un bon début

On peut **prouver** qu'un langage n'est **PAS** régulier, par exemple avec le « lemme de l'Étoile »

Exemple typique : **$L = \{a^n b^n, n > 0\}$ n'est pas régulier**

TRAITEMENTS SÉMANTIQUES

Reconnaître un mot c'est bien,
savoir quoi en faire c'est mieux !

TRAITEMENTS SÉMANTIQUES

Attention : on sort du cadre de la FORME pure, pour prendre en compte des éléments de SENS.

Plan de ce qui va suivre :

Préalable : programmation événementielle

Traitements associés à un automates d'états fini
Ex : calcul d'un nombre entier, autocommutateur

[PS : De façon analogue, nous reparlerons de traitements sémantiques dans le cadre des grammaires formelles]

PROGRAMMATION PAR RÈGLES, PROGRAMMATION ÉVÉNEMENTIELLE

La programmation impérative n'est qu'une forme (paradigme) parmi d'autres de styles de programmation : programmation logique, fonctionnelle, déclarative, événementielle, temps réel...

Programmation événementielle : on associe un traitement à un événement

Programmation par règles :

le traitement d'une règle peut être exécuté dès que les prémisses de cette règle sont vérifiées ;

l'ordre d'exécution des règles dépend donc essentiellement de l'ordre de validation des prémisses
toutefois, si plusieurs prémisses sont vérifiées simultanément, le système peut prendre en compte l'ordre des règles pour choisir laquelle exécuter.

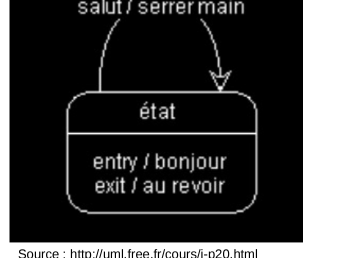
TRAITEMENTS SÉMANTIQUES D'UN AEF

On peut associer des traitements sémantiques à un automates :

Sur ses transitions

Dans ses états

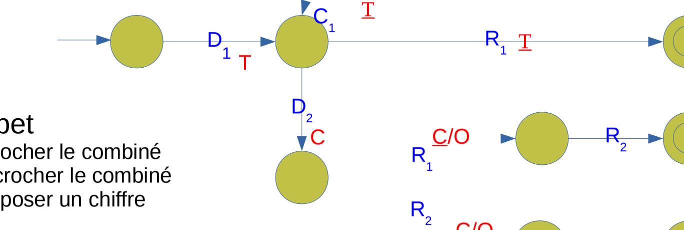
Exemple : diagrammes d'états-transitions



Source : http://uml.free.fr/cours/i-p20.html

EXEMPLE : RETOUR SUR L'AUTOCOMMUTATEUR

Traitements associés uniquement aux transitions



Alphabet
D : Décrocher le combiné
R : Raccrocher le combiné
C : Composer un chiffre

Traitements :
T : émettre tonalité (T : la couper)
C : établir la communication (C : la couper)
O : émettre « occupé » (O : la couper)

EXEMPLE 2 : RECONNAISSANCE ET CALCUL D'UN NOMBRE ENTIER

La forme d'un nombre entier est facile à définir

Comment en dériver une valeur numérique correspondante ?

combien vaut « 456 » en base 10 ?

et dans une autre base ?

On peut associer des calculs aux transitions de l'automate

VALIDITÉ DE L'APPROCHE

S'appuyer sur un AEF pour gérer des traitements, c'est très pratique.

Mais ce n'est pas prévu dans le formalisme initial

Quid des belles propriétés de ce formalisme (par ex, algo de déterminisation) ?

Quid des traitements en cas de « retours arrière »/« backtracking » (cf AEF non déterministes) ?

=> Démarche : ne s'appuyer que sur des AEF Déterministes [éventuellement complets]

GRAMMAIRES FORMELLES

Ensemble de règles pour générer (et analyser) les mots (phrases) d'un langage

DÉFINITION

Définition : une grammaire G est définie par un quadruplet <X, V, P, S> où

X : ensemble fini de symboles, l'**alphabet terminal**

V : ensemble fini de symboles **non-terminaux**

S : élément de V appelé **axiome** de départ

P : ensemble de **règles de production**, de la forme $\alpha \to \beta$, avec $\alpha \in (X \cup V)^*.V.(X \cup V)^*$ et $\beta \in (X \cup V)^*$

Exemple :
V={S,A,B}, X={a,b}, axiome S,
P={S→A, S→bB, bB→bbB, A→aA, A→ε, bB→ε}

EX : JAVA LANGUAGE SPECIFICATION

3.7 Comments
There are two kinds of comments:

`/* text */` A traditional comment: all the text from the ASCII characters /* to the ASCII characters */ is ignored (as in C and C++).
`// text` A end-of-line comment: all the text from the ASCII characters // to the end of the line is ignored (as in C++).

These comments are formally specified by the following productions:

```
Comment:
    TraditionalComment
    EndOfLineComment

TraditionalComment:
    /* NotStar CommentTail

EndOfLineComment:
    // CharactersInLineopt LineTerminator

CommentTail:
    * CommentTailStar
    NotStar CommentTail

CommentTailStar:
    /
    * CommentTailStar
    NotStarNotSlash CommentTail

NotStar:
    ...

NotStarNotSlash:
    ...
```

Non terminaux

Régle de production

Terminaux

These productions imply all of the following properties:

- Comments do not nest.
- /* and */ have no special meaning in comments that begin with //.
- // has no special meaning in comments that begin with /* or /**.

As a result, the text:
/* this comment /* // /** ends here: */
is a single complete comment.

The lexical grammar implies that comments do not occur within character literals (§3.10.4) or string literals (§3.10.5).

GÉNÉRATION D'UN MOT PAR UNE GRAMMAIRE

Déf : u ∈ (X∪V)* se *dérive directement* en v ∈ (X∪V)* par G ssi u=xαy, v=xβy et α→β ∈ P. On note u→v

Déf : u se *dérive* en v ssi il existe une chaîne de dérivation directe menant de u à v. On note u→*v.

V={S,A,B}, X={a,b}, axiome S,
P={S→A, S→bB, bB→bbB, A→aA, A→ε, bB→ε}

S →A→ aA→ aaA → aaε → aa *c-à-d* S →* aa

Déf : Le langage engendré par G<X,V,P,S> est défini par $L_G(S)$={u ∈ X* / S →*u}

Déf : deux grammaires sont équivalentes si elles engendrent le même langage.

HIÉRARCHIE DES GRAMMAIRES

Idée (N. Chomsky - 1956) : classer les langages en fonction de leur complexité syntaxique ; s'appuyer sur les grammaires.

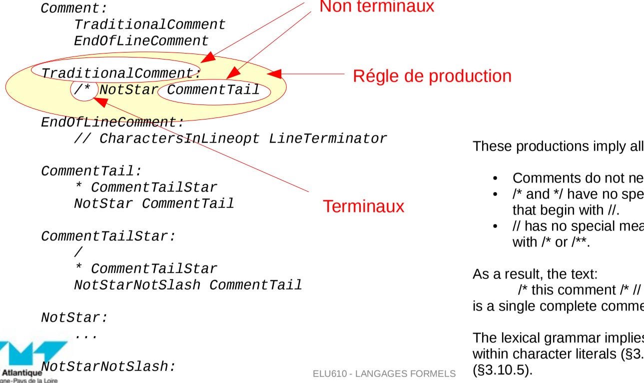
Type 0 : aucune restriction sur P : grammaires générales
α→β, avec α∈(X∪V)*.V.(X∪V)* et β∈(X∪V)*

Type 1 (grammaires contextuelles)
$u_1 A u_2 \to u_1 m u_2$, avec u1, u2 ∈(X∪V)*, A∈ V, m ∈(X∪V)*

Type 2 (grammaires hors-contexte / algébriques)
A→m, avec A∈ V, m ∈(X∪V)*

Type 3 (grammaires régulières)
A→m, avec A∈ V, m ∈(X*.V)∪X* - grammaire linéaire à droite
ou
A→m, avec A∈ V, m ∈(V.X*)∪X* - grammaire linéaire à gauche

Par construction, type 3 ⊂ type 2 ⊂ type 1 ⊂ type 0



HIÉRARCHIE DES GRAMMAIRES (2)

Chaque classe grammaticale peut être « traitée » par un outillage spécifique

Générales (Type 0) – machine de Turing

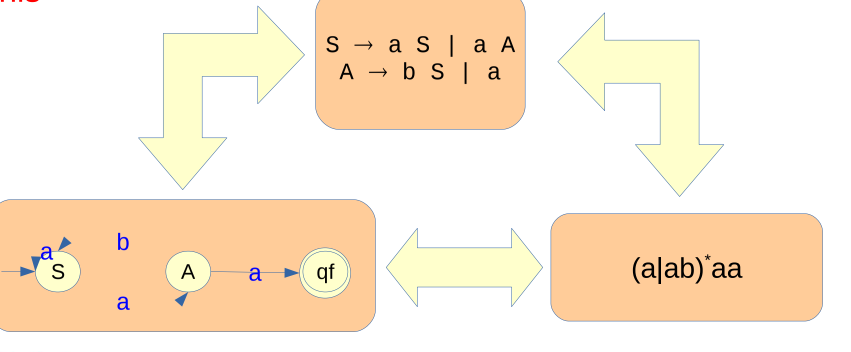
Contextuelles (Type 1) – sous catégorie de machines de Turing (ressources limitées)

Hors-contexte (type 2) ou algébrique – automates à pile

Rationnelle (type 3) – automates d'états finis

LANGAGES RÉGULIERS 70

Il y a équivalence entre les formalismes de grammaire régulière, d'expressions régulières et automates d'états finis



LANGAGES/GRAMMAIRES HORS-CONTEXTE 71

Seulement un non-terminal en partie gauche des règles de production

A→m, avec A∈ V, m ∈(X∪V)*

Contre-exemple :

V={S,A,B}, X={a,b}, axiome S,

P={S→A, S→bB, bB→bbB, A→aA, A→ε, bB→ε}

Exemple :

V={S,A,B}, X={a,b}, axiome S,

P={**S**→A, **S**→bB, **B**→bbB, **A**→aA, **A**→ε, **B**→ε}

S →bB→ bbbB→ bbbε → bbb *(c-à-d S →* bbb)*

Lecture : http://fr.wikipedia.org/wiki/Grammaire_non_contextuelle

ARBRE SYNTAXIQUE 72

Rappel : Le mot appartient au langage ssi il existe une dérivation menant de l'axiome à ce mot

ex : S →A→ aA→ aaA → aaε → aa

Dans le cas d'une grammaire hors-contexte, la dérivation peut se représenter sous forme bi-dimensionnelle : c'est l'arbre [de dérivation] syntaxique

S — Racine=axiome

A — Noeud= non-terminal

a A

a A — Dérivation directe

ε — Feuille=terminal

APPARTENANCE D'UN MOT À UN LANGAGES HC 73

Décidabilité :

Construire toutes les dérivations de longueur <= |mot|*2

Temps de calcul exponentiel

Autre algo en $O(n^3)$

Utilisabilité :

Pour une sous-classe de langages hors-contexte, on sait faire en O(n) : LL(n), LR(n), LALR(n),...

EXEMPLE : EXPRESSIONS ARITHMÉTIQUES

Grammaire : E → E+E | E−E | E*E | E/E | (E) | nb

a − b * c ?

Ambiguïté !

E→ E-E

E→ nb

E→ E*E

E→ nb

E→ nb

???

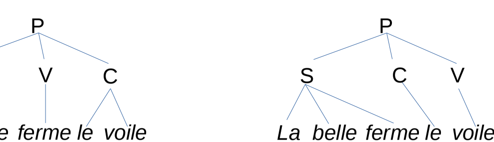
a – b * c

a – b * c

AMBIGUÏTÉ

Définition : une grammaire est **ambiguë** lorsqu'**il existe** plusieurs arbres de dérivation différents pour un même mot du langage

Analogie : une phrase ambiguë



L'ambiguïté est une mauvaise propriété pour une grammaire
- elle induit un comportement non prédictible de la machine

2 solutions pour supprimer ce problème :
- Transformation de la grammaire
- Directives supplémentaires de certains outils

SUPPRESSION DE L'AMBIGUÏTÉ

Nous allons (juste) CONSTATER qu'une modification de la grammaire peut supprimer l'ambiguïté
- L'identification des modifications à apporter peut être difficile, et est en tous cas hors des objectifs de ce cours

Il s'agit juste d'un EXEMPLE

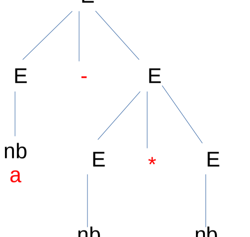
Bien sûr, si le langage est intrinsèquement ambigu (cf ex précédent), on ne peut résoudre proprement le problème.

SUPPRESSION DE L'AMBIGUÏTÉ (1)

Grammaire de base :

E → E + E | E – E | E * E | E / E | (E) | nb

a - b * c



-> Cette grammaire ne prend pas en compte
les conventions d'écriture liées aux priorités des opérateurs

SUPPRESSION DE L'AMBIGUÏTÉ (2) — 78

Grammaire de base

E → E + E | E – E | E * E | E / E | (E) | nb

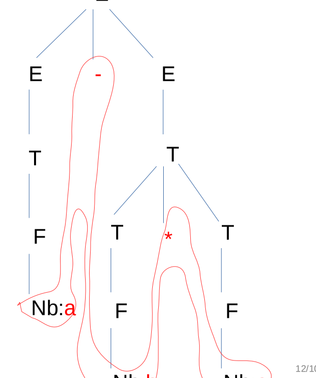
Prise en compte des priorités :

E → E + E | E – E | T

T → T * T | T / T | F

F → nb | (E)

a - b * c



SUPPRESSION DE L'AMBIGUÏTÉ (3) — 79

Grammaire de base

E → E + E | E – E | E * E | E / E | (E) | nb

Prise en compte des priorités

E → E + E | E – E | T

T → T * T | T / T | F

F → nb | (E)

a - b * c

a – b – c ?

SUPPRESSION DE L'AMBIGUÏTÉ (4) — 80

Grammaire de base :

E → E + E | E – E | E * E | E / E | (E) | nb

Prise en compte des priorités :

E → E + E | E – E | T

T → T * T | T / T | F

F → nb | (E)

Explicitation de l'associativité

E → E + T | E – T | T

T → T * F | T / F | F

F → nb | (E)

a – b – c

SUPPRESSION DE L'AMBIGUÏTÉ (5) — 81

Modifier la grammaire nous a donc permis de supprimer les ambiguïtés.

Les problèmes d'ambiguïté rencontrés dans notre exemple (priorité, associativité) sont des problèmes typiques, à tel point que les outils de manipulation de grammaires comportent souvent des directives spécifiques pour pallier à ces ambiguïtés.

Par ex, yacc permet de déclarer directement les priorités des symboles avec les directives spécifiques %left, %right, %nonassoc :

E → E + E | E – E | E * E | E / E | (E) | nb

AUTRES PROPRIÉTÉS DES GRAMMAIRES

On distingue d'autres propriétés intéressantes : pré-réduite, réduite, propre.

Les outils de manipulation des grammaires peuvent détecter que la grammaire qu'on leur fournit n'a pas ces propriétés (idem que pour ambiguïté).

Ils savent même transformer la grammaire pour qu'elle ait ces propriétés (en supprimant les règles de production inutiles).

Malheureusement, il ne faut pas laisser faire l'outil sans se poser de question : la plupart du temps, la bonne approche consiste à RAJOUTER des règles, pas à en enlever. Ce qui doit être fait par l'utilisateur, pas par l'outil.

=> ATTENTION aux messages d'erreurs des outils

TRAITEMENTS SÉMANTIQUES (2)

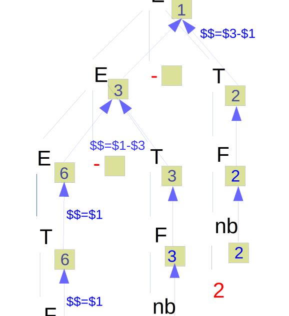
Comme pour les automates, il est possible de s'appuyer sur le formalisme des grammaires pour piloter des traitements sémantiques

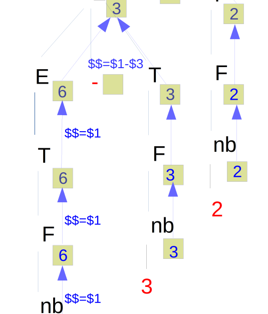
GRAMMAIRE : EXPRESSION ARITHMÉTIQUE

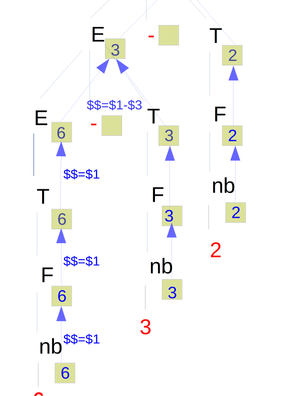
E → E + T {$$ = $1 + $3;}
  | E – T {$$ = $1 - $3;}
  | T {$$ = $1;}

T → T * F {$$ = $1 * $3;}
  | T / F {$$ = $1 / $3;}
  | F {$$ = $1;}

F → nb {$$ = $1;}
  | (E) {$$ = $2;}



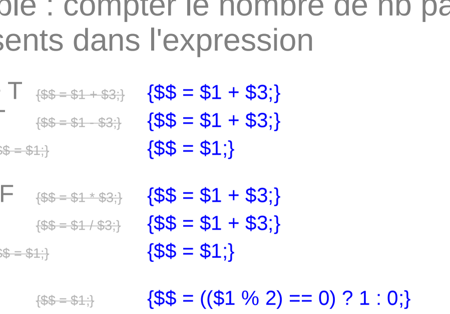




DÉCOUPLAGE SYNTAXE / SÉMANTIQUE

On peut facilement changer la sémantique.

Exemple : compter le nombre de nb pairs présents dans l'expression



E → E + T ~~{$$ = $1 + $3;}~~ {$$ = $1 + $3;}
  | E – T ~~{$$ = $1 – $3;}~~ {$$ = $1 + $3;}
  | T ~~{$$ = $1;}~~ {$$ = $1;}

T → T * F ~~{$$ = $1 * $3;}~~ {$$ = $1 + $3;}
  | T / F ~~{$$ = $1 / $3;}~~ {$$ = $1 + $3;}
  | F ~~{$$ = $1;}~~ {$$ = $1;}

F → nb ~~{$$ = $1;}~~ {$$ = (($1 % 2) == 0) ? 1 : 0;}
  | (E) ~~{$$ = $2;}~~ {$$ = $2;}

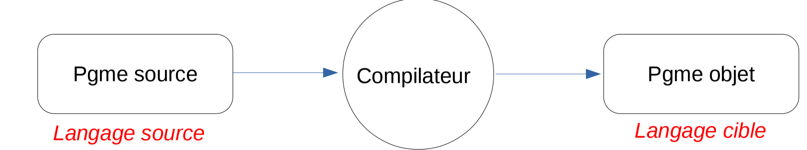
PRINCIPALES PHASES DE LA COMPILATION

Des principes, des outils, des pratiques

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

QU'EST-CE QU'UN COMPILATEUR ? 87

Compilation : traduction d'un programme écrit en langage source (évolué) en un programme équivalent en langage cible (souvent langage « machine », mais pas toujours).



« Accessoirement », détection et explicitation des erreurs (→ aide au programmeur)

PRINCIPALES ÉTAPES DE LA COMPILATION 88

Découpage en couches → évolutivité, réutilisabilité

Frontal (« front-end ») : dépend du langage source

Vérifier que le programme source respecte les règles du langage source (le programme source appartient-il au langage source ?)

Élaborer une représentation intermédiaire du programme source (arbre syntaxique, TDS, ...) adaptée aux traitements ultérieurs

Dorsal (« back-end ») : dépend du langage cible

générer le code objet, sur la base du travail effectué par le frontal

COMPILATION – LE FRONTAL 89

Analyse lexicale : reconnaître les « mots » dans la chaîne de caractères que constitue le programme source.
Tous les « mots » utilisés sont-ils autorisés ?

Analyse syntaxique : reconnaître la structuration générale du programme source (arbre syntaxique)
Le programme source est-il « correctement » écrit ?
Génération arbre syntaxique (abstrait)

Contrôles sémantiques :
Vérifications statiques complémentaires (typage par ex)
Gestion d'une « table des symboles »

ANALYSE LEXICALE

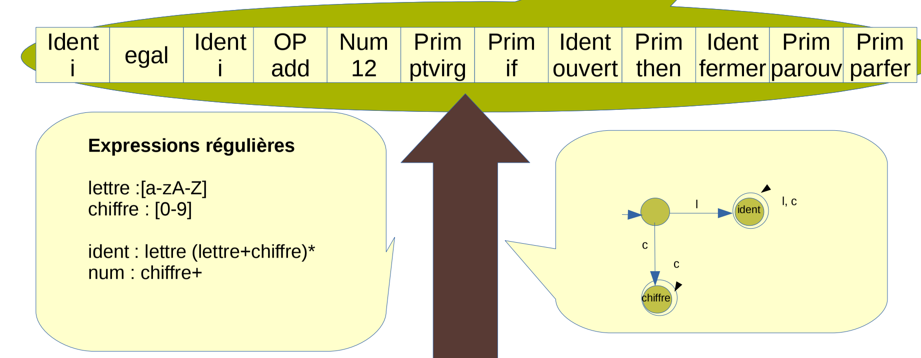
Filtrer le flot de caractères d'entrée et reconnaître les entités lexicales (tokens / lexèmes)

- Sauter les commentaires
- Mots-clés, identificateurs, nombres, ...
- Calculs de bas niveau (par ex. nombres)

Définition des lexèmes : expressions régulières

Mode opératoire préférentiel : modèle d'automate d'états fini + traitements

ANALYSE LEXICALE - EXEMPLE



ANALYSE SYNTAXIQUE

Reconstruction de la structure globale du programme

- Arbre syntaxique ; arbre de syntaxe abstraite

Symboles terminaux = lexèmes

- Partage du travail avec l'analyse lexicale

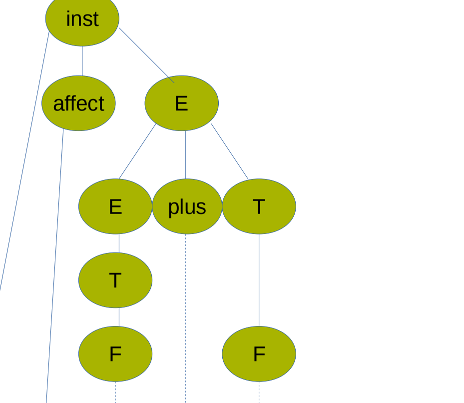
Mode opératoire :

- Analyse ascendante (LR) vs descendante (LL)
- Automate à pile

Typologie des grammaires et des analyseurs

- LL(n), LR(n), LALR(n), ...

ANALYSE SYNTAXIQUE - EXEMPLE



...
Inst : ident affect E
...
E : E plus T
...
E : T
...
T : F
...
F : ident
F : num

ANALYSE SÉMANTIQUE

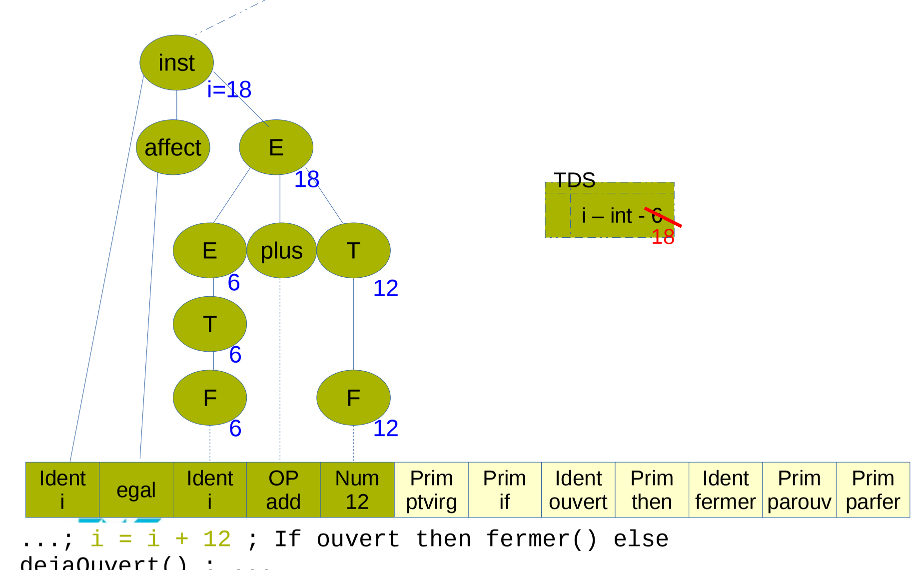
On ne peut pas tout vérifier dans le cadre de l'analyse syntaxique

Grammaire attribuée

Table des symboles

Typage

VUE GLOBALE - INTERPRÉTATION



...; i = i + 12 ; If ouvert then fermer() else dejaOuvert() ; ...

VUE GLOBALE - COMPILATION

inst
mov i,1
addIm 1, 12
mov 1, i
affect
E
mov i, 1
addIm 1, 12
E
plus
T
mov i, 1
12
T
mov i, 1
F
mov i, 1
F
12

Gestion des symboles
Gestion des registres
Gestion mémoire

Ident i	egal	Ident i	OP add	Num 12	Prim ptvirg	Prim if	Ident ouvert	Prim then	Ident fermer	Prim parouv	Prim parfer	…

...; i = i + 12 ; If ouvert then fermer() else dejaOuvert() ; …

COMPILATION – LE DORSAL

Génération code intermédiaire
Meilleure portabilité

Génération code cible
Spécifique de la machine cible

Optimisation de code
Parfois très coûteuse
90% du temps d'exécution correspond à 10% du code
Non souhaitable pour le debug => option spécifique

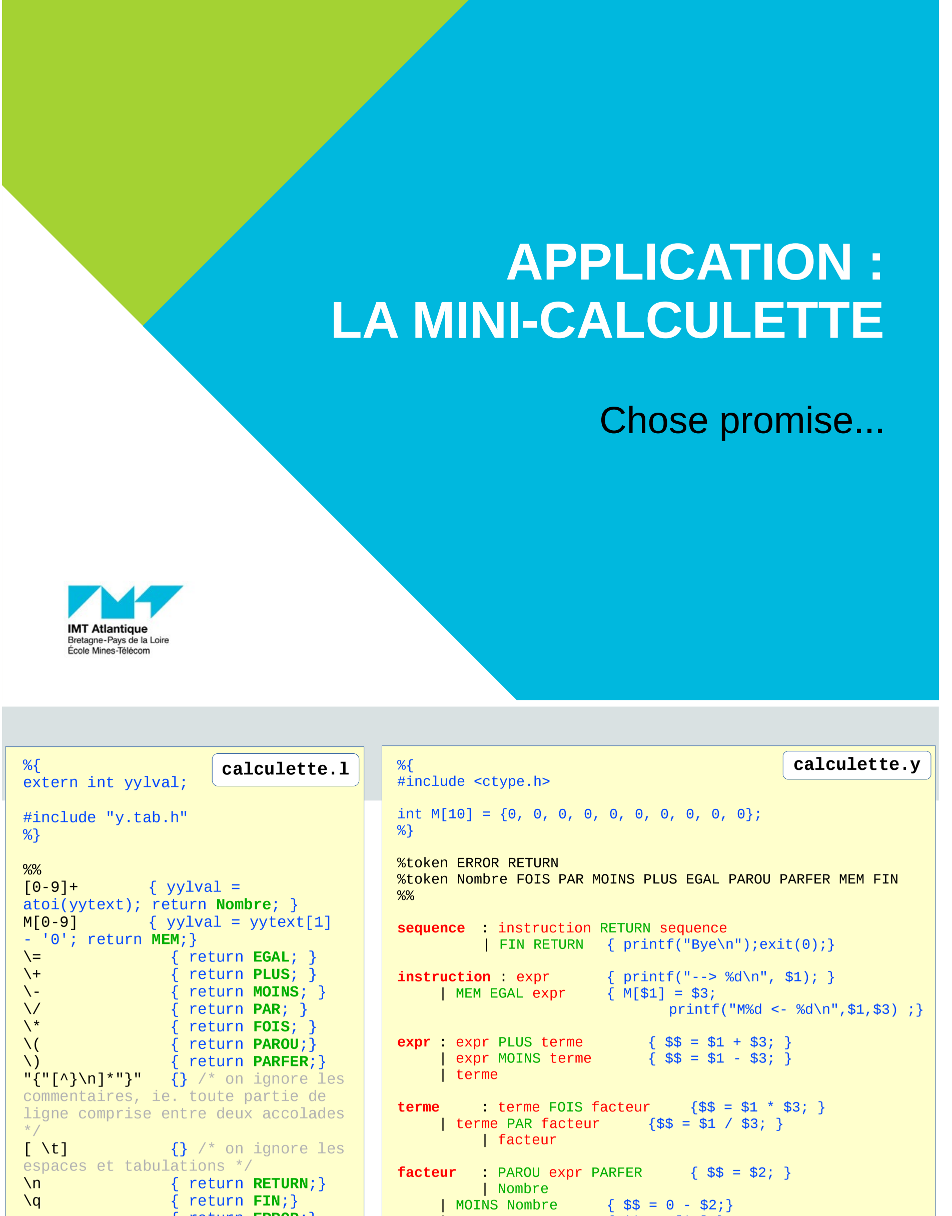
Pas abordé dans le cadre de ce cours

OUTILS – GÉNÉRATEURS DE COMPILATEURS

Principe : on définit le strict minimum et l'outil génère le code du compilateur

Lex, Flex, Ocamllex, ..: analyse lexicale

Yacc, Bison, Ocamlyacc : analyse syntaxique



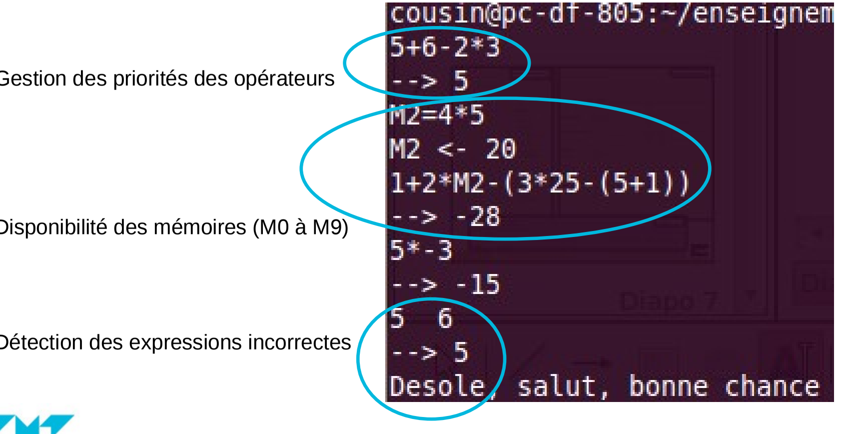
RAPPEL

Travail de la mini-calculette :

Gestion des priorités des opérateurs

Disponibilité des mémoires (M0 à M9)

Détection des expressions incorrectes



calculette.l

```
%{
extern int yylval;

#include "y.tab.h"
%}

%%
[0-9]+         { yylval =
atoi(yytext); return Nombre; }
M[0-9]         { yylval = yytext[1]
- '0'; return MEM;}
\=                 { return EGAL; }
\+                 { return PLUS; }
\-                 { return MOINS; }
\/                 { return PAR; }
\*                 { return FOIS; }
\(                 { return PAROU;}
\)                 { return PARFER;}
"{"[^}\n]*"}"      {} /* on ignore les
commentaires, ie. toute partie de
ligne comprise entre deux accolades
*/
[ \t]              {} /* on ignore les
espaces et tabulations */
\n                 { return RETURN;}
\q                 { return FIN;}
.                  { return ERROR;}
%%
```

calculette.y

```
%{
#include <ctype.h>

int M[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
%}

%token ERROR RETURN
%token Nombre FOIS PAR MOINS PLUS EGAL PAROU PARFER MEM FIN
%%

sequence   : instruction RETURN sequence
           | FIN RETURN   { printf("Bye\n");exit(0);}

instruction : expr        { printf("--> %d\n", $1); }
     | MEM EGAL expr      { M[$1] = $3;
                            printf("M%d <- %d\n",$1,$3) ;}

expr : expr PLUS terme        { $$ = $1 + $3; }
     | expr MOINS terme       { $$ = $1 - $3; }
     | terme

terme      : terme FOIS facteur     {$$ = $1 * $3; }
     | terme PAR facteur      {$$ = $1 / $3; }
           | facteur

facteur    : PAROU expr PARFER      { $$ = $2; }
           | Nombre
     | MOINS Nombre       { $$ = 0 - $2;}
     | MEM                { $$ = M[$1];}
%%
```

main.c

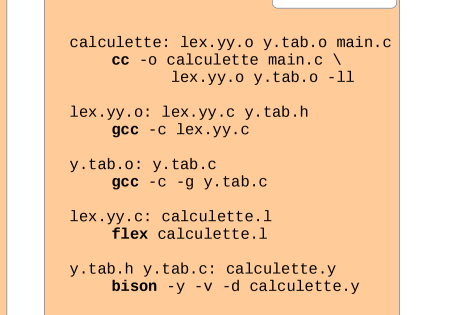
```
#include <stdio.h>
extern int yylineno;
extern char  yytext[];

yyerror() {printf( "Erreur ligne %d, symbole: %s\n", yylineno, yytext); exit(1);}

main() {  yyparse(); printf("\nJ'ai fini!\n");}
```

MINI-CALCULETTE : GÉNÉRATION

102



makefile

```
calculette: lex.yy.o y.tab.o main.c
     cc -o calculette main.c \
          lex.yy.o y.tab.o -ll

lex.yy.o: lex.yy.c y.tab.h
     gcc -c lex.yy.c

y.tab.o: y.tab.c
     gcc -c -g y.tab.c

lex.yy.c: calculette.l
     flex calculette.l

y.tab.h y.tab.c: calculette.y
     bison -y -v -d calculette.y
```


MOTS CLÉS

103

Compilateur / Interpréteur

Machine virtuelle

Lexical / Syntaxique / Sémantique

Arbre syntaxique, syntaxe abstraite

Grammaire formelle

Automate

Conflits, ambiguïté

Table des symboles

Langage source / langage cible

BIBLIOGRAPHIE

104

Modern Compiler implementation in [Java/C/ML] - A. Appel (Cambridge University Press)

Crafting a compiler - C. Fischer, R. LeBlanc, R.K. Cytron (Addison Wesley Longman).

Compilateurs – D. Grune, H.E. Bal, C. Jacobs, K. Langendoen (Dunod)

The theory of parsing, translation, and compiling - A. V. Aho, J.D. Ullman

Wikipedia.org :

http://fr.wikipedia.org/wiki/Compilation_(informatique)

http://fr.wikipedia.org/wiki/Expression_régulière

http://fr.wikipedia.org/wiki/Analyse_syntaxique

http://fr.wikipedia.org/wiki/Grammaire_formelle

ELU610 - TP Expressions régulières

OBJECTIFS :

Le but de ce TP est de vous donner un aperçu de la puissance des expressions régulières. Nous allons pour cela utiliser leur implémentation dans le langage Java.

APPLICATION SIMPLE : `String.matches(String)`

1. Rendez-vous sur la page https://regex101.com/ qui vous permet de tester interactivement des expressions régulières. Élaborez une expression régulière qui caractérise la forme d'une adresse de courrier électronique. Vous disposez en annexe d'un résumé des notations utilisables dans une expression régulière.

2. La classe `TesterPattern.java` - disponible sur votre espace Moodle – lit des caractères entrés au clavier et indique si chaque chaîne entrée par l'utilisateur ressemble à un motif défini par une expression régulière. Elle utilise pour cela méthode `matches()` de la classe `String`.

```
import java.util.Scanner;
public class TesterPattern {

    public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      String regex = "test"; // expression régulière à rechercher
      String chaine; // chaîne à traiter
      do{
         chaine=sc.nextLine();
         if (chaine.matches(regex)) System.out.println(chaine + "--> OK");
         else System.out.println(chaine + "--> PAS OK");
      } while (sc.hasNext());
    }
}
```

Introduisez l'expression régulière de l'exercice précédent dans `TesterPattern.java` pour tester si la chaîne entrée ressemble à une *adresse électronique*.

3. De la même manière, tester la conformité avec le motif `(a*ba*)*` de la chaîne suivante : `ababaababababaaababababababbaaaaababababababababababbababababababc`
Que remarquez-vous ?

JAVA : UTILISATION DU PACKAGE `regex`

Pour utiliser les expressions régulières en Java il faut importer `java.util.regex` :

```
import java.util.regex.* ;
```

Une expression régulière est fournie sous forme de `String`. Ainsi, par exemple, "`a*b|c`" est une expression régulière qui définit le langage contenant le mot `c` et les mots formés d'un nombre quelconque de `a` et finissant par un `b`.

Pour être recherchée, une expression régulière doit d'abord être « compilée » et le résultat stocké dans un objet `Pattern` :

```
Pattern p = Pattern.compile ("a*b|c");
```
[1]

Ce `Pattern` va ensuite être appliqué à la chaîne de caractères à traiter. Ceci donne un `Matcher` :

```
String entree = "aabbbcab"; // la chaîne à traiter
Matcher m = p.matcher(entree);
```

La méthode `find()` permet de trouver successivement tous les appariements du `Matcher`. Les méthodes `start()` et `end()` donnent l'index de début et de fin de l'appariement dans la chaîne d'entrée :

```
while (m.find())
   System.out.println(entree.substring(m.start(), m.end()));
```

EXERCICE

Écrivez et testez une classe contenant une fonction `main` qui réalise les actions précédentes.

1 On remarque ici que `compile` est une méthode de classe

EXERCICES - TRAVAIL AVEC LES EXPRESSIONS RÉGULIÈRES ET LES GROUPES

Après cette courte introduction, nous allons travailler sur le fichier `gen1551.csv` qui contient des données sur des personnes : nom, prénom, date et lieu de naissance, date et lieux de mariage, nom et prénom du conjoint, date et lieux du décès, … Ce fichier est de type `csv`, c'est-à-dire un fichier de caractères comportant des lignes contenant chacune des données séparées par des points-virgules.

Outre ce fichier, vous récupérerez sur Moodle le code Java ci-après qui permet de le lire ligne par ligne, d'appliquer un traitement identique à chacune des lignes, et d'afficher le résultat de ce traitement le cas échéant :

```
public class Genealogie {
  public static void main (String[] args){
    String leFichier = "../gen1551.csv"; // Adresse du fichier
    String uneLigne;
    String uneLigneTraitee;
    try {
      BufferedReader input =
            new BufferedReader (new FileReader (leFichier));
      while (input.ready()) {
        uneLigne = input.readLine();
        uneLigneTraitee = traiter(uneLigne);
        if (!uneLigneTraitee.equals(""))
            System.out.println(uneLigneTraitee);
      }
      input.close();
    }
    catch (FileNotFoundException ex) {
          System.out.println("Fichier inexistant !!");
        }
      catch (IOException ex) {
          System.out.println("Erreur de lecture du fichier !!");
        }
  }

  public static String traiter (String ligne){
    // Mettre ici le traitement souhaité
    return(ligne);
  }
}
```

Votre travail de TP consistera à écrire différentes variantes de la fonction de traitement pour obtenir l'effet indiqué.

PRÉPARATION

Ouvrez le fichier `gen1551.csv` en mode texte (avec `gedit` par exemple) et observez sa structure et les données qu'il contient. Vous pouvez en parallèle ouvrir ce fichier avec un tableur pour accéder plus facilement aux données, mais vous n'y verrez pas les caractères séparateurs point-virgules.

TRAITEMENT 1

Que fait le programme si on utilise le traitement suivant ?

```
public static String traiter1 (String ligne){
     Pattern p = Pattern.compile("^([0-9]+);[^;]*;PAUL;");
     Matcher m = p.matcher(ligne);
     String res="";
     if (m.find())
       res=(m.group(1) + "   " + ligne);
     return res;
```

Traitement 2

Complétez ce traitement afin de ne garder que les gens nés dans un village dont le nom commence par `PLOU`.

Traitement 3

Modifiez le traitement pour que *l'affichage* des lieux de naissance des personnes trouvées dans l'exercice 2 change le début `PLOU` en `LOC` (par exemple `PLOUNEVEZ` devient `LOCNEVEZ`).

Traitement 4

Incrémentez de 10 ans les dates de naissance affichées des personnes trouvées dans l'exercice 2.

Traitement 5

Compter le nombre de fiches où le nom de la personne est `ABALAIN`.

Annexe - Syntaxe des expressions régulières «à la Perl»

Voici un aperçu d'une partie de la syntaxe des expressions régulières «à la Perl» :

- `toto` va trouver les sous-chaînes `toto` ;
- `.` est un caractère quelconque, mis à part le passage à la ligne `\n` et le retour chariot `\r` ;
- `[ax123Z]` signifie : « un caractère quelconque parmi `a`, `x`, `1`, `2`, `3` et `Z` » ;
- `[A-Z]` signifie : « un caractère quelconque dans l'intervalle de `A` à `Z` » ;
- le trait d'union sert à indiquer les intervalles mais peut faire partie des caractères recherchés s'il est placé à la fin : `[AZ-]` signifie : « un caractère quelconque parmi `A`, `Z` et `-` » ;
- on peut combiner à volonté les caractères énumérés et les intervalles. Par exemple `[A-Za-z0-9.:?]` signifie « une lettre majuscule ou minuscule, un chiffre, un point, un deux-points, ou un point d'interrogation » ;
- les caractères `(`, `)`, `\`, `[`, `]` peuvent être recherchés, à condition de les protéger par un antislash : `\(`, `\)`, `\\`, `\[`, `\]` ;
- le symbole `^` placé *après le crochet ouvrant* indique que l'on va chercher le complémentaire de ce qui est placé entre les crochets. Exemple : `[^a-z]` va trouver un caractère quelconque qui ne soit *pas* une lettre entre `a` et `z` ;
- on dispose des *quantificateurs* suivants : `*` (zéro, une ou plusieurs fois), `+` (une ou plusieurs fois), `?` (zéro ou une fois), `{n,m}` (entre n et m fois), `{n,}` (au moins n fois) ;
- on dispose également de versions « non gourmands » de ces quantificateurs : `*?`, `+?`, `??`, `{n,m}?`, `{n,}?`

  La différence entre quantificateurs « gourmands » et « non gourmands » provient du fait que les premiers vont trouver la sous-chaîne la plus longue respectant les contraintes alors que les deuxièmes vont trouver la chaîne la plus courte.

  Exemple : l'expression `[a-z]+` appliquée à « `mon ami Pierrot` » va trouver `mon` alors que `[a-z]+?` va trouver `m` (ce qui n'a que peu d'intérêt). Autre exemple (qui montre l'utilité des quantificateurs non gourmands) : l'expression `\(.+\)` appliquée à « `Brest (29) et Aix (13)` » va retourner `(29) et Aix (13)` puisque c'est la plus longue sous-chaîne délimitée par une parenthèse ouvrante et une parenthèse fermante. Par contre `\(.+?\)` va retourner d'abord `(29)` et ensuite `(13)` ;
- les symboles `^` et `$` servent à indiquer le début et la fin d'une chaîne. Par exemple : `^a.+` va trouver toutes les chaînes qui *commencent* par un `a`, `toto$` va trouver toutes les chaînes qui *finissent* par `toto`, `^ $` va trouver toutes les chaînes égales à un blanc ;
- l'opérateur « ou » `|` sert à indiquer un choix entre deux expressions ;
- on peut utiliser les parenthèses pour deux raisons :

  1. pour délimiter une expression qui sera utilisée par l'opérateur « ou » ou à laquelle on va appliquer un quantificateur (exemple : `abc(toto)+` signifie « `abc` suivi d'un ou plusieurs `toto` ») ;
  2. pour définir un « groupe », c'est-à-dire une sous-chaîne que l'on va récupérer par la suite.

  Ce double usage des parenthèses peut être gênant : en écrivant `abc(toto)+` on fait de `toto` un groupe, même si on n'a pas l'intention de le récupérer par la suite. Ce n'est pas très grave en soi, mais si on le souhaite on peut explicitement empêcher la création du groupe en écrivant `abc(? :toto)+`

ELU610 - TP Automates Etats Finis

OBJECTIFS PÉDAGOGIQUES

Le but de ce TP est de comprendre comment passer du formalisme des automates à une implémentation logicielle concrète. Ceci permet de bien assimiler le principe de *reconnaissance des mots par un automate*, et de mettre en évidence les implications pratiques du *déterminisme*. Nous mettrons avant tout l'accent sur les aspects algorithmiques.

NB : Le travail demandé est ici basé sur une programmation en langage Java, mais on pourra tout aussi bien le réaliser dans un autre langage orienté objet, par exemple Python.

INTRODUCTION

Nous devons trouver un moyen de simuler le travail d'un automate : lorsqu'on proposera une chaîne de caractères en entrée, ce simulateur devra répondre `true` si l'automate en question accepte/reconnaît cette chaîne, et `false` sinon. Rappelons qu'un automate accepte une chaîne si et seulement si il a un chemin correspondant au mot et menant d'un état initial à un état terminal.

Par exemple, l'automate suivant accepte la chaîne `ababa`, mais pas `abab`.

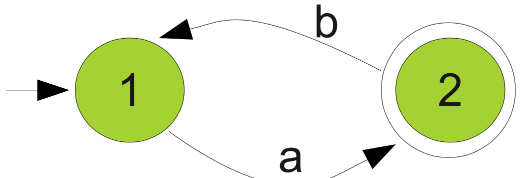


Illustration 1: Aef1, un automate simple

CONSIGNES PRÉALABLES

Chaque automate sera simulé par une classe dérivée de la classe abstraite `Aef` :

```
public abstract class Aef {
        public abstract boolean accepte(String input);
}
```

Pour tester interactivement les différents automates que nous simulerons, nous pourrons utiliser un programme principal dont l'objet sera de soumettre interactivement des chaînes de caractères à l'automate désiré, celui-ci étant une instance d'une classe fille de `Aef` :

```
import java.util.*;
public class Simu {
  public static void main(String[] args) {
    Aef aef = new Aef1(); // l'automate a simuler
    Scanner scan = new Scanner(System.in);
    while (true) {
      System.out.println("Veuillez entrer votre chaine de test");
      String input = scan.next();
      if (aef.accepte(input)) System.out.println("Chaine " + input + " acceptee");
      else System.out.println ("Chaine " + input + " refusee");
}  } }
```

Voyons maintenant les principaux éléments de notre simulateur d'automate.

GESTION DE LA CHAÎNE D'ENTRÉE

Pour s'exécuter, l'automate va lire la chaîne d'entrée caractère par caractère. Une approche impérative classique consiste à travailler directement sur la chaîne avec un index pour connaître la position du caractère courant.

Quelles méthodes de la classe `String` permettront de savoir s'il reste des caractères à traiter, connaître le caractère courant, et connaître le reste de la chaîne à traiter ?

SIMULATION D'UN AUTOMATE DÉTERMINISTE

La simulation d'un automate déterministe est simple. En partant de son *état de départ* et en suivant à chaque itération la *transition* concernée par le *caractère lu*, trois situations peuvent être rencontrées au final :

- On a lu tout le mot et on se retrouve dans un *état terminal* ; le mot est alors accepté.
- On a lu tout le mot et on se retrouve dans un *état non terminal* ; le mot est refusé.
- On ne peut pas lire tout le mot car le caractère lu n'est pas prévu pour l'état courant (c-à-d il n'y a pas de transition correspondant à la configuration rencontrée) ; le mot est alors également refusé.

Voici un exemple de mise en œuvre de `Aef1` correspondant à l'automate donné par l'Illustration 1. Pour simplifier, les états de l'automate sont représentés par des entiers.

```
public class Aef1 extends Aef{
  /**
   * Teste si une chaîne est acceptée par l'automate simulé
   * Version impérative
   * @param entree - la chaîne de caractères a tester
   * @return - true si la chaîne est acceptée, false sinon
   */
  public boolean accepte (String entree){
    int etat = 1; // état initial de l'automate
    int index = 0; // rang du premier caractère a traiter
    char carlu; // caractère courant

    while (index != entree.length()) {
    // tant qu'il reste des caractères a traiter
      carlu = entree.charAt(index++); // lecture caractère courant et passage
au suivant
      if ((etat == 1)&&(carlu == 'a')) etat = 2;
      else if ((etat == 2)&&(carlu=='b')) etat=1;
      else return false ; // si aucune transition, entree n'est pas acceptée
    }
    // il n'y a plus rien a lire : est-on dans un état terminal ?
    if (etat==2) return true; // entree est acceptée
    else return false; // entree n'est pas acceptée
}}
```

Avec le programme principal, testez interactivement cette réalisation sur des chaînes appartenant au langage, et sur d'autres n'appartenant pas au langage.[1]

1 Rappel : pour compiler/exécuter une classe `MaClasse` définie dans un package `monPackage`, le fichier `MaClasse.java` doit être stocké dans un répertoire `monPackage`, et les commande de compilation (`javac`) ou d'exécution (`java`) doivent être lancées depuis le répertoire père.

SIMULATION D'UN DEUXIÈME AUTOMATE DÉTERMINISTE

On veut maintenant simuler un autre automate :

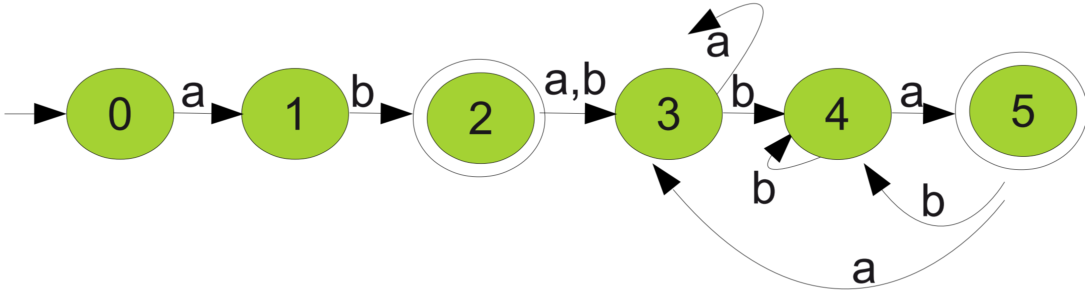


Illustration 2: Aef2, un autre automate déterministe

Caractérisez en *langue naturelle* le langage reconnu par cet automate. Donnez quelques exemples et contre-exemples de mots.

Créez une classe `Aef2` correspondant à cet automate, et testez-la.

Que pensez-vous des codes `Aef1` et `Aef2` ? Quelles critiques pouvez-vous en faire ?

TABLE DE TRANSITION

Plutôt que de diluer les spécificités de chaque automate dans des lignes de code, nous allons maintenant utiliser des tables de transition. Celles-ci récapitulent toutes les transitions prévues dans l'automate et permettent typiquement de savoir, à partir d'un état courant et d'un symbole lu, quel sera le prochain état. Voici ce que cela donne pour `Aef2` :

Aef2	a	b
0	1	.
1	.	2
2	3	3
3	3	4
4	5	4
5	3	4

ITÉRATION N°1

Dans une transposition Java, chaque automate pourra donc disposer d'une méthode `transition()` qui joue le rôle de table de transition en renvoyant pour chaque configuration l'état destination :

```
private int transition (int etatCourant, char symboleLu){...}
```

Comment représenter l'absence de transition ?

De même, une méthode `isTerminal()` permettra de savoir si un état est terminal :

```
private boolean isTerminal (int etat){...}
```

et `initial()` renverra l'(unique) état initial de l'automate :

```
private int initial (){...}
```

Créez une nouvelle version `Aef2tt` de l'automate `Aef2` en utilisant un tel schéma.

Itération n°2

Que pensez-vous de la réutilisabilité de ce nouveau code ? Dans l'hypothèse où l'on souhaite développer plusieurs automates différents, peut-on factoriser plus de code entre leurs implémentations respectives ? Essayez de maximiser cette réutilisabilité en mettant tout le code possible dans la classe mère `Aef`, et testez en réécrivant une nouvelle version de l'automate `Aef1` que vous pourrez appeler `Aeftt1`.

Itération n°3

Plutôt que d'être un code (verbeux) prévoyant tous les cas possibles pour un automate en particulier, la méthode `transition()` pourrait être un code plus général (et synthétique) qui accède aux configurations possibles dans une structure de données adéquate. Quelle(s) structure(s) de données pourrait-on utiliser en Java pour représenter la table de transition ?

Élaborez sur ce principe une nouvelle version de `Aef2` que vous appellerez `Aef2ttBis`.

Quel impact y-a-t-il sur la lisibilité du code ? En quoi ce concept de table de transition pourrait-il être utile à des logiciels – tel que l'algorithme de déterminisation - qui produisent de nouveaux automates ?

POUR ALLER PLUS LOIN

Traitements sémantiques

Des traitements sémantiques peuvent être associés aux transitions de l'automate. Chaque traversée d'une transition provoque alors l'exécution du traitement associé, *qui peut être différent selon les transitions*.

Ces traitements peuvent être représentés par des objets `Runnable` dont l'exécution sera déclenchée au moment adéquat :

```
class Ts1 implements Runnable {
  public void run(){...un traitement sémantique...}
}
class Ts2 implements Runnable {
  public void run(){...un autre traitement sémantique...}
}
...
Runnable ts1 = new Ts1();
Runnable ts2 = new Ts2();
...
ts1.run();
...
ts2.run();
...
```

Modifiez votre classe `Aef2tt` pour associer le traitement « affichage d'un point » à chaque transition de votre automate, ainsi qu'un traitement « afficher 'Etat Terminal' » à la transition 4→5 avec `a`. Testez. Peut-on prévoir le nombre de points affichés ?

Que se passe-t-il si on associe des traitements sémantiques aux transitions d'un automate non déterministe ?

Simulation d'un automate non déterministe sans transition sur le mot vide

On repart de votre code sans prise en compte des traitements sémantiques. Soit `Aefnd2` une version non

déterministe de `Aef2` (cf Illustration 3).

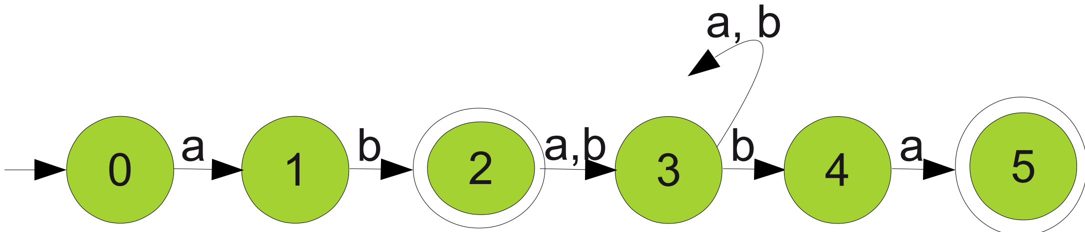


Illustration 3: Aefnd2, une version non déterministe de Aef2

Pourquoi `Aefnd2` n'est-il pas déterministe ?

Que faut-il modifier dans votre code pour pouvoir simuler un tel automate, et de façon générale pour pouvoir simuler un automate non déterministe sans transition sur le mot vide ? Quel devrait être l'impact sur la lisibilité du code et sur ses performances ?

En vous inspirant des classes `Aef` et `AefX` que vous avez déjà développées, écrivez une classe `Aefnd` et sa classe fille `Aefnd2` qui permettent de simuler `Aefnd2`. Testez-les avec des cas judicieux.

VERS LA PRISE EN COMPTE DES TRANSITIONS SUR LE MOT VIDE

Soit `Aefnd3` une version de `Aefnd2` dans laquelle on a rajouté quelques transitions sur le mot vide (cf Illustration 4). Comment pourrait-on simuler un tel automate ? Quels sont les problèmes spécifiques à résoudre [2]? Quelles sont les conséquences sur la lisibilité du code et sur ses performances ?

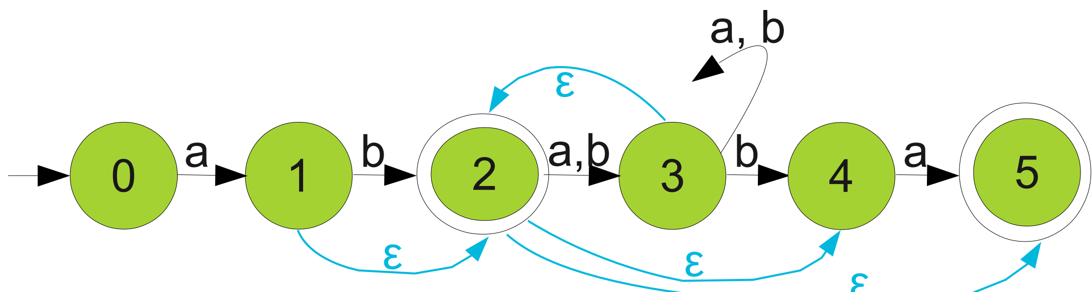


Illustration 4: Aefnd3, un automate non déterministe avec des transitions sur le mot vide

2 Indice : voir la définition de ce qu'est la fermeture transitive d'un graphe.

ELU610/PC1 : Automates d'états finis, expressions régulières, grammaires formelles

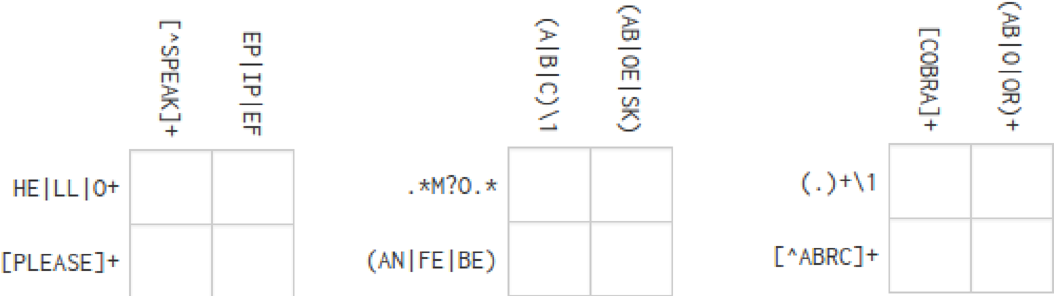
OBJECTIFS D'APPRENTISSAGE

Voir objectifs rappelés dans le document d'auto-évaluation

EXERCICE 1 – EXPRESSIONS RÉGULIÈRES

(tiré de http://regexcrossword.com)

Trouver quel symbole mettre dans chaque case pour respecter les définitions fournies en horizontal et vertical.



EXERCICE 2

Soit le langage $L_1 = \{a^n b^{2m+1}, n \geq 0, m \geq 0\}$.

1. Donnez une *expression régulière,* un *automate d'états fini* et une *grammaire formelle* qui lui correspondent.
2. Les mots suivants sont-ils acceptés par votre automate (justifiez votre réponse) : `abb`, `bbb`, `aabbb` . Qu'en concluez-vous ?
3. Donnez la table de transition de votre automate. Est-il déterministe ? complet ?
4. En vous appuyant sur votre grammaire, indiquez (et justifiez) si les mots suivants appartiennent au langage : `abb`, `bbb`, `aabbb`
5. Que pensez-vous du cas où on veut imposer `n=m` : $L1' = \{a^n b^{2n+1}, n \geq 0\}$?

EXERCICE 3

Soit L_2 le langage des mots w sur {a, b} tels que |w|>5 et dont les deux dernières lettres sont les mêmes que les deux premières, mais en ordre inverse.

1. Faire pour L_2 le même travail que dans l'exercice 2, avec les mots `aababaa` et `babab`

2. Le cas échéant, trouvez un automate équivalent au vôtre qui soit *déterministe complet*. Les mots `aababaa` et `babab` sont-ils acceptés par ce nouvel automate ? Est-ce étonnant ?

3. Rajoutez à ce nouvel automate des traitements sémantiques permettant de calculer la différence entre le nombre de `a` et le nombre de `b` présents dans le mot.

4. Rajoutez à votre grammaire les traitements sémantiques permettant de faire la même chose. Décorez un ou deux arbres syntaxiques de votre choix avec les valeurs d'attributs correspondant à ce traitement.

ELU610/PC SUPPLÉMENTAIRE : AUTOMATES D'ÉTATS FINIS, EXPRESSIONS RÉGULIÈRES ET GRAMMAIRES FORMELLES

EXERCICES SUPPLÉMENTAIRES (POUR S'ENTRAÎNER HORS SÉANCE !)

EXERCICE A

Soit le langage L_3= { (a + b)* a a b (a + b)*}

1. Décrivez L_3 par une phrase en langage naturel
2. Donnez un automate d'états fini reconnaissant L_3
3. Cet automate est-il déterministe ? Justifiez votre réponse.
4. Le cas échéant, essayez de trouver un automate équivalent qui soit ***déterministe complet***

EXERCICE B

Soit L_7 le langage des constantes numériques en base 7.

Une telle constante numérique comprend une partie entière, éventuellement suivie d'un point et d'une partie décimale. La partie entière comme la partie décimale sont constituées d'un ou plusieurs chiffres, compris entre 0 et 6. La présence de zéros non significatifs (ie. à gauche de la partie entière ou à droite de la partie décimale) est interdite.

Exemples : 502 123.456 0.12 1.0 3.0002 0.0 0

Contre-exemples : 7 02 2.00

- Donnez une expression régulière définissant L_7.
- Donnez un automate d'état fini déterministe reconnaissant L_7.
- Dotez votre automate de traitements sémantiques permettant d'évaluer les constantes numériques. A la fin du traitement, les deux variables prédéfinies `ent` et `dec` devront valoir respectivement la valeur de la partie entière et la valeur de la partie décimale (0 s'il n'y en a pas).

EXERCICE C

On considère les mots sur l'alphabet X={a, b, c}.

Donner un AEF déterministe qui reconnaisse tous les mots contenant au moins une des sous-chaines 'a b c' ou 'b c b'.

EXERCICE D

On rappelle ci-dessous la spécification d'un identificateur telle qu'elle est fournie dans le manuel de

référence Caml :

“Identifiers are sequence of letters, digits and _ (the underscore character), starting with a letter. Letters contain a least the 52 lowercase and uppercase letter from the ASCII set. Implementations can recognise as letters other characters from the extended ASCII set. Identifiers cannot contain two adjacent underscore characters (__).”

Donner une expression régulière sur l’alphabet X = L ∪ C ∪ U définissant les identificateurs Caml ; L représente l’ensemble des lettres (éventuellement accentuées), C l’ensemble des chiffres et U le symbole underscore (U = {_}).

EXERCICE E

Le langage L = {entiers naturels pairs écrits en base 10} est-il régulier ? Justifiez votre réponse.

EXERCICE F

Soit R une expression régulière quelconque. Not(R) représente l’ensemble de toutes les chaînes qui n’appartiennent pas à l’ensemble régulier défini par R. Montrer que Not(R) est un ensemble régulier (Idée : transformer l’automate reconnaissant R en un automate reconnaissant Not(R)).

Application : montrer que l’ensemble des chaînes sur X={`a`, `b`} qui ne contiennent pas trois `b` consécutifs forme un langage régulier.

EXERCICE G

Voici la définition syntaxique des constantes numériques flottantes du langage de programmation Caml telle que fournie dans le manuel de référence du langage :

Les constantes numériques flottantes sont constituées d’une partie entière, une partie décimale et une partie exposant.
La partie entière est une séquence d’un ou plusieurs chiffres, éventuellement précédée d’un signe moins. La partie décimale consiste en un point suivi de zéro , un ou plusieurs chiffres. L’exposant est constitué du caractère e ou E, d’un signe optionnel et enfin d’une suite de un ou plusieurs chiffres.
La partie décimale ou la partie exposant peuvent être omises, mais pas les deux, ceci pour éviter les ambiguités avec les constantes numériques entières.
Exemples : 5467. -67.34 -4.23E5 1.0008E-4 -5.e2
Contre-exemples : 345 .56 5E +4.3

1. Construisez un automate d’états fini qui reconnaisse ce langage. Pour alléger les notations, on utilisera le symbole terminal `ch` pour représenter un chiffre (`0|1|...|9`). Donnez également sa **table de transition**.

2. Donnez une spécification de ce langage sous forme d’expression régulière ; on pourra remplacer le symbole du méta-langage “+” (ou) par le symbole “|” afin d’éviter toute confusion avec le symbole “+” des nombres signés.

ELU610 - TP Grammaires formelles

Objectifs d'apprentissage

A travers l'utilisation d'un « compilateur de compilateur » sur un exemple pédagogique simple, ce TP doit vous amener à être capable de :

- élaborer une grammaire hors-contexte simple
- expliquer la problématique d'ambiguïté d'une grammaire formelle
- expliquer le positionnement des phases d'analyse lexicale, syntaxique et sémantique

La calculette

Nous reprenons ici l'exemple de la calculette vue en cours. Il s'agit d'élaborer une calculette interactive, c'est-à-dire un logiciel capable d'interpréter interactivement des expressions arithmétiques entrées par l'utilisateur. Pour ce faire nous allons notamment utiliser un générateur d'analyseur lexical (`flex`) et un générateur de compilateur (`bison`) qui vont nous permettre de nous concentrer sur la définition des lexèmes et de la grammaire formelle.

Infrastructure du logiciel et outils mobilisés

Récupérez sur Moodle le squelette d'application mis à votre disposition. Il contient :

- `calculette.l` : quelques définitions lexicales, que vous allez compléter
- `calculette.y` : quelques définitions syntaxiques, que vous allez compléter
- `main.c` : programme principal
- un fichier makefile permettant de générer l'application.

Générez la version de départ de l'application : `make -f make_calc`

Testez-la avec quelques expressions simples :

```
$ ./calculette
5*5
--> 25
5*5+5
--> 30
<ctl-D>
J'ai fini!
```

A noter que ce programme lit les expressions interactivement dans son `stdin`. On peut donc également lui faire traiter un fichier contenant des expressions arithmétiques : `./calculette < batch`

Dans cette version de départ, seules l'addition et la multiplication d'entiers sont prévues. Il vous appartient maintenant de modifier[1] `calculette.l` et `calculette.y` pour permettre aussi la soustraction et la division.

1 Sans (trop) tricher sur le support de cours...

LES NOMBRES ROMAINS

En vous inspirant de la calculette, réaliser un logiciel interactif qui affiche pour chaque nombre entré en numération romaine (cf https://fr.wikipedia.org/wiki/Numération_romaine) sa valeur en base 10. Pour commencer, vous restreindrez la liste des lexèmes aux symboles utilisables dans un nombre romain à : `I`, `V`, `X`. N'hésitez pas à dessiner quelques *arbres syntaxiques* pour vous aider à élaborer votre grammaire.

Voici un exemple d'utilisation d'un tel logiciel :

```
$ ./romains
i
Valeur : 1
v
Valeur : 5
x
Valeur : 10
xxiv
Valeur : 24
<ctl-D>
J'ai fini!
```

POUR LES PLUS GOURMANDS

Vous avez aimé les exercices précédents, en voici d'autres pour affiner votre maîtrise. Ils sont indépendants et peuvent donc être réalisés dans n'importe quel ordre (et selon vos goûts).

CALCULETTE ROMAINE

Combinez le travail des deux exercices précédents pour créer la première calculette interactive travaillant avec des nombres en numération romaine. Pour simplifier, l'affichage des résultats restera en base 10.[2]

CALCULETTE « SCIENTIFIQUE »

Parmi les améliorations possibles de la calculette :

- Permettre l'utilisation de nombres « réels » (et pas seulement des entiers naturels).
- Autoriser d'autres opérateurs : '`-`' unaire, mise au carré, `log`, `exp`, … tout est permis !
- Ajouter une(des) mémoire(s) à votre calculatrice.

2 Contrairement à la *lecture* d'un nombre romain qui s'inscrit pleinement dans la thématique des langages formels, *l'écriture* d'une valeur en numération romaine ne pourra pas tirer bénéfice des techniques de cette thématique et est une question purement algorithmique. Cela n'a donc pas d'intérêt pédagogique dans le cadre du présent TP.

ELU610/AUTO-ÉVALUATION

Ce petit mémo pourra vous aider à vous positionner vis à vis des objectifs d'apprentissage de la partie « Langes formels » de l'UE ELU610.

RÉSUMÉ DES COMPÉTENCES ATTENDUES

Les objectifs du modules, énoncés dans la fiche programme ou rappelés dans l'espace Moodle du module donnent déjà pas mal d'information. Il s'agit tout d'abord de connaître et savoir utiliser à bon escient la *terminologie du domaine* :

- expression régulière
- automate d'états fini
- automate déterministe, non déterministe
- grammaire formelle, grammaire régulière, grammaire hors-contexte
- grammaire ambiguë
- analyse lexicale, syntaxique, traitement sémantique
- arbre [de dérivation] syntaxique

Sur la base des activités menées dans l'UE, chaque élève devrait par ailleurs être capable de :

- définir un langage simple par une expression régulière
- élaborer un automate d'états fini correspondant à un langage simple
- utiliser un automate – qu'il soit déterministe ou pas - pour reconnaître un mot
- élaborer une grammaire correspondant à un langage
- trouver des arbres de dérivation correspondant à des mots simples pour une grammaire formelle imposée
- comprendre et respecter une spécification formelle (expression régulière et/ou grammaire) de langage
- adjoindre des traitements sémantiques à un automate d'états fini ou à une grammaire

EXERCICE TYPE

Voici plus en détail le travail typiquement demandé dans le cadre d'une évaluation finale.

NB : « typiquement » ne veut pas dire « systématiquement ».

Soit L un langage défini de manière informelle.

1. Donner une expressions régulière définissant L.

   Il s'agit ici de savoir utiliser les expressions régulières. Il sera primordial de s'assurer que a) tous les mots du langage sont bien conformes à l'expression et b) tous les mots conformes à l'expression appartiennent bien au langage. Ces deux conditions sont nécessaires, puisqu'elle permettent de statuer a) sur l'inclusion du langage L dans le langage défini par l'expression régulière et b) l'inclusion réciproque.

La « simplicité » de l'expression régulière fournie est un critère moins fondamental : on n'attend pas une solution « optimale », mais pour le moins il faut démontrer la maîtrise des constructions disponibles et éviter trop de complication.

2. Trouver un AEF reconnaissant L ;

   On retrouve ici la même démarche et les mêmes préoccupations que celles énoncées précédemment. Une version déterministe de l'automate pourra éventuellement être demandée. La maîtrise de l'algorithme de déterminisation n'est pas demandée : ce qui compte, c'est de savoir « créer » un automate déterministe correspondant à un langage donné.

3. Le tester sur quelques mots appartenant à L et quelques mots n'appartenant pas à L.

   Il faut justifier clairement l'acceptation ou la non-acceptation des mots, sur la base du fonctionnement de l'automate et non de la définition du langage. Pour montrer qu'un mot est accepté, il faut donc identifier clairement le chemin correspondant au mot dans l'automate, chemin qui doit partir d'un état initial et finir dans un état d'arrivée, et ces informations doivent être explicitées. Dans le cas de la non-acceptation par un automate non déterministe, on veillera à ce que tous les chemins possibles aient bien été envisagés avant de conclure à la non-acceptation.

4. Expliquer/illustrer la problématique du déterminisme.

   Il s'agit ici de commenter/justifier le comportement algorithmique de la reconnaissance d'un mot: linéaire si déterministe, exponentiel sinon.

5. Trouver une grammaire correspondant à L

   Même remarques que pour expressions régulières et automates.

6. Donner un arbre de dérivation pour un mot de L

   Cette « preuve » de l'appartenance d'un mot au langage engendré par la grammaire nécessite pour le moins que les dérivations utilisées dans la dérivation correspondent bien aux règles de production de la grammaire (il ne faut pas en inventer sur mesure !)

Voir en PC1 des exemples de langages à traiter :

- L1 = {an b2m+1, n ≥ 0, m ≥ 0}
- mots w sur {a, b} tels que |w|>5 et dont les deux dernières

Troisième partie

λ-calculus, functional programming, compilation, typing, OCaml

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

The λ-calculus

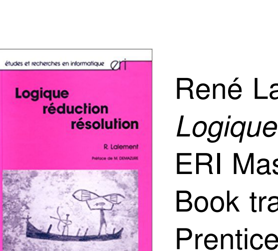
Mathematical modeling of functions

Fabien Dagnat
ELU 610 – C5
1st semester 2019

1. The syntactic landscape
2. Computing with syntactic objects
3. Conclusion

The λ-calculus [Church]

- A formal language proposed by Alonzo Church in the 1930s to model the notion of function
- http://en.wikipedia.org/wiki/Lambda_calculus
- We will use it to
  - illustrate the notion of formal language
  - understand fundamentals of formal reasoning
  - introduce the functional paradigm



René Lalement
Logique Réduction Résolution
ERI Masson, 1990
Book translated in english *Computation as logic*,
Prentice-Hall in 1993, ISBN 9780137700097

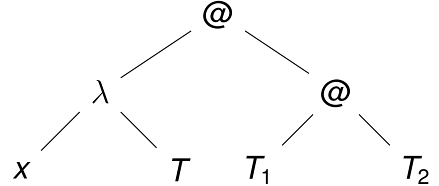
1. The syntactic landscape
2. Computing with syntactic objects
3. Conclusion

The syntax of the λ-calculus

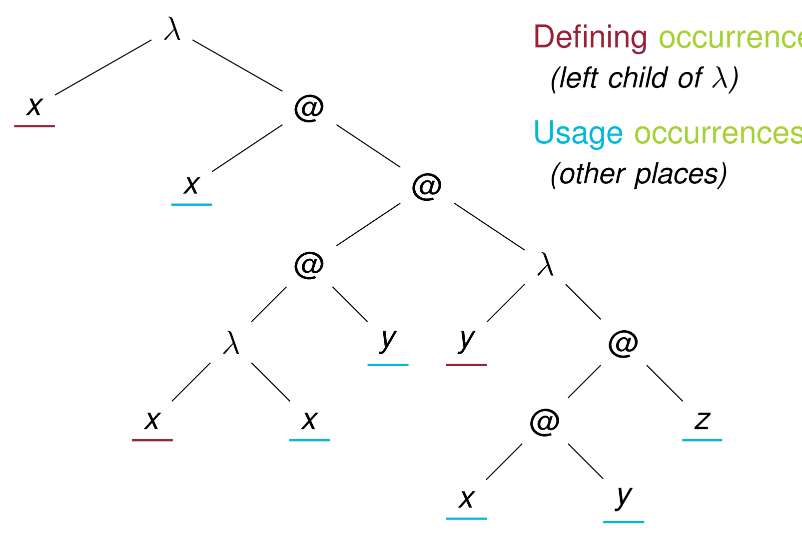
- The set $\Lambda_{\mathcal{X}}$ of the terms defined by
  - variables x, y, ... from a denumerable set $\mathcal{X}$
  - applications $(T_1 T_2)$ of a term T_1 (the function) to a term T_2 (the argument)
  - functions $(\lambda x.T)$ of a variable x (the parameter) and a term T (the body)
- BNF: $T ::= x \mid (TT) \mid (\lambda x.T)$
- Parenthesis may be omitted
  - outer: $(T_1 T_2) = T_1 T_2$ and $(\lambda x.T) = \lambda x.T$
  - application is left associative: $T_1 T_2 T_3 = (T_1 T_2) T_3$
  - λ is right associative: $\lambda x.\lambda y.T = \lambda x.(\lambda y.T)$ and $\lambda x.T_1 T_2 = \lambda x.(T_1 T_2)$
- Some well-known λ-terms
  - $\lambda x.x = \mathbf{I}$ $\quad$ $\lambda x.\lambda y.x = \mathbf{K}$ $\quad$ $\lambda x.\lambda y.\lambda z.((xz)(yz)) = \mathbf{S}$

Abstract syntax

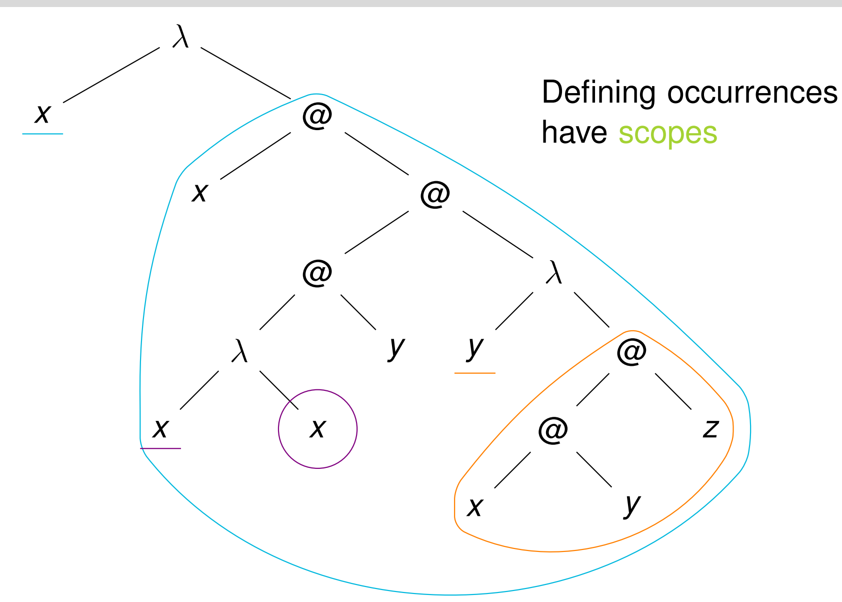
- $\Lambda_{\mathcal{X}} = T_{\{@,\lambda\}}[\mathcal{X}]$ with
  - @ is the only constructor and $ar(@) = 2$
  - λ is the only binder and $ar(\lambda) = 1$
- Terms are trees
  - variables are leaves
  - constructors and binders are nodes
- ex: $(\lambda x.T)(T_1 T_2)$



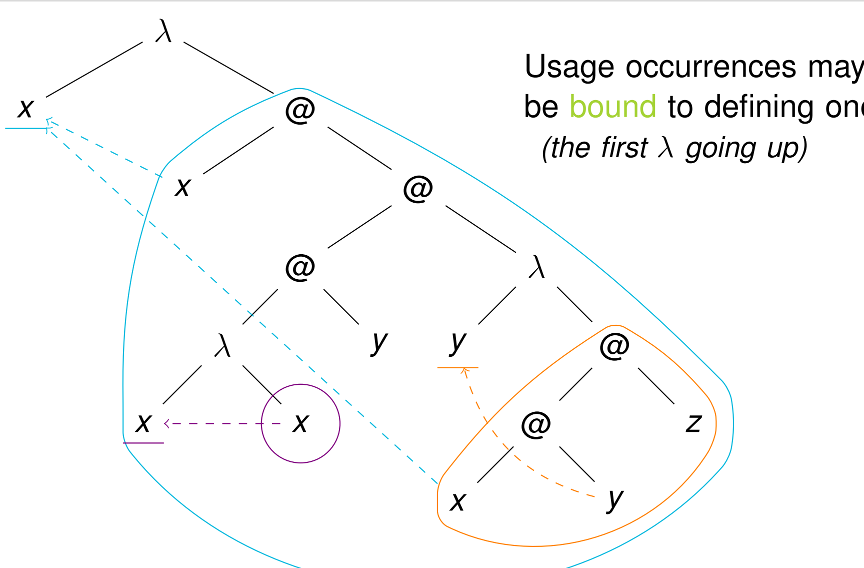
Variables, scope and binding



Variables, scope and binding



Variables, scope and binding

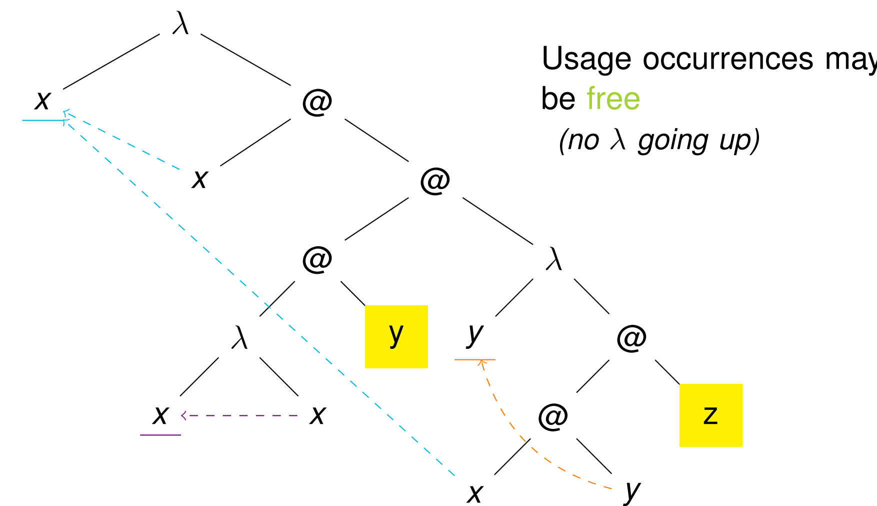


Usage occurrences may be bound to defining ones

(the first λ going up)

Variables, scope and binding

7 / 26



Usage occurrences may be free

(no λ going up)

Free and bound variables

- A free variable is defined outside the term
  - a kind of *global variable* (for the term)
  - its name is essential and cannot be modified
  - $\lambda x.y$ is different from $\lambda x.z$
- A bound variable is intern to the term
  - a kind of *local variable* (for the term)
  - its name can be modified (the defining occurrence and all its depending bound occurrences)
  - $\lambda x.x$ is identical to $\lambda y.y$
  - known as α-conversion (see later for the mathematical definition)
  - the name of a bound variable has no importance, only the link to its binder[1]
- A term with free variables is open
- A term with no free variables is closed (*a.k.a.* combinators)

[1] there exists notations without names, see for example [Bou08]

Reminder

- One can define a function f on $\mathbb{N}$ recursively by
  1. defining $f(0)$
  2. defining $f(n+1)$ in terms of $f(n)$

  for example, factorial
  1. $0! = 1$
  2. $(n+1)! = (n+1)n!$
- One can prove a property P on $\mathbb{N}$ by
  1. proving $P(0)$
  2. proving that if $P(n)$ holds, $P(n+1)$ is true

  for example, if $P(n)$ is $0+1+\cdots+n = \frac{n(n+1)}{2}$
  1. $0 = 0$
  2. $0+1+\cdots+n+(n+1) = \frac{n(n+1)}{2} + (n+1) = (n+1)(\frac{n}{2}+1)$
     $= \frac{(n+1)(n+2)}{2}$
- Variants: starting at k or $P(0), \ldots, P(n) \Rightarrow P(n+1)$

Induction I

- (Structural) induction is a method of definition or proof on the set of terms $T_\Sigma[\mathcal{X}]$
- One can define a function f on $\Lambda_\mathcal{X}$ inductively by
  1. defining f on $\mathcal{X}$ (leaves)
  2. defining $f(T_1 T_2)$ in terms of $f(T_1)$ and $f(T_2)$
  3. defining $f(\lambda x.T)$ in terms of $f(T)$
- For example, the set of free variables FV is defined by
  1. $FV(x) = \{x\}$
  2. $FV(T_1 T_2) = FV(T_1) \cup FV(T_2)$
  3. $FV(\lambda x.T) = FV(T) \setminus \{x\}$
- For example, the size of a λ-term is defined by
  1. $\texttt{size}(x) = 1$
  2. $\texttt{size}(T_1 T_2) = \texttt{size}(T_1) + \texttt{size}(T_2) + 1$
  3. $\texttt{size}(\lambda x.T) = \texttt{size}(T) + 1$

Induction II

- One can prove a property P on $\Lambda_\mathcal{X}$ inductively by
  1. proving P on $\mathcal{X}$
  2. proving $P(T_1 T_2)$ supposing $P(T_1)$ and $P(T_2)$ are true
  3. proving $P(\lambda x.T)$ supposing $P(T)$ is true
- Prove $\forall T \in \Lambda_\mathcal{X}, card(FV(T)) \leq \texttt{size}(T)$
  1. $card(FV(x)) = card(\{x\}) = 1 = \texttt{size}(x)$
  2. let's suppose $card(FV(T_i)) \leq \texttt{size}(T_i)$ for i in $\{1,2\}$ (IH)

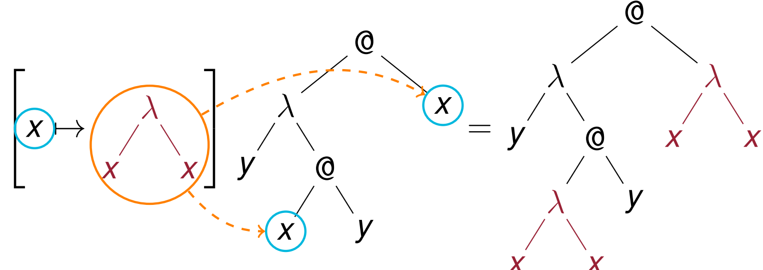
$$\begin{aligned} card(FV(T_1 T_2)) &= card(FV(T_1) \cup FV(T_2)) && \text{def of } FV \\ &\leq card(FV(T_1)) + card(FV(T_2)) && \text{prop of } card \\ &\leq \texttt{size}(T_1) + \texttt{size}(T_2) && \text{IH} \\ &\leq \texttt{size}(T_1 T_2) && \text{def of } \texttt{size} \end{aligned}$$

  3. let's suppose $card(FV(T)) \leq \texttt{size}(T)$ (IH)

$$\begin{aligned} card(FV(\lambda x.T)) &= card(FV(T) \setminus \{x\}) && \text{def of } FV \\ &\leq card(FV(T)) && \text{prop of } card \\ &\leq \texttt{size}(T) && \text{IH} \\ &\leq \texttt{size}(\lambda x.T) && \text{def of } \texttt{size} \end{aligned}$$

Substitutions

- Giving a meaning to a free variable is done by substitution
- Substitution is a function associating a term to
  - a variable (the substituted variable) and
  - two terms (the replacement term and the term on which substitution operates)
- $[x \mapsto T_1]T_2$ is the term defined by replacing **all** free occurrences of x within T_2 by T_1



Substitution on $\Lambda_\mathcal{X}$

- Defined inductively

$$\begin{cases} [x \mapsto T]x = T & \\ [x \mapsto T]y = y & \text{if } x \neq y \\ [x \mapsto T]T_1 T_2 = [x \mapsto T]T_1 [x \mapsto T]T_2 & \\ [x \mapsto T]\lambda y.T' = \lambda y.[x \mapsto T]T' & \text{if } x \neq y, y \notin FV(T) \end{cases}$$

  the last condition prevent captures of a free y in T
- The definition is incomplete *e.g.* $[x \mapsto T]\lambda x.T'$, $[x \mapsto y]\lambda y.T$
- α-conversion (a.k.a. α-equivalence) is defined by
  $\lambda x.T =_\alpha \lambda y.[x \mapsto y]T$ if $y \notin FV(T)$ (*freshness condition*)
- The definition of substitution is complete modulo renaming
  - if $x = y$ or $y \in FV(M)$, we rename the bound y
- We always work on $\Lambda_\mathcal{X}/{=_\alpha}$ (modulo renaming)

$$\begin{cases} (1)\ [x \mapsto T]x = T & \\ (2)\ [x \mapsto T]y = y & \text{if } x \neq y \\ (3)\ [x \mapsto T]T_1T_2 = [x \mapsto T]T_1[x \mapsto T]T_2 & \\ (4)\ [x \mapsto T]\lambda y.T' = \lambda y.[x \mapsto T]T' & \text{if } x \neq y \text{ and } y \notin FV(T) \\ & \\ (\alpha)\ \lambda x.T = \lambda y.[x \mapsto y]T & \text{if } y \notin FV(T) \end{cases}$$

- $[z \mapsto \lambda x.xy]\lambda z.x\ (\lambda y.zy) =$

$$\begin{aligned} &{}^{(3)} &&= [z \mapsto \lambda x.xy]\lambda z.x\ ([z \mapsto \lambda x.xy]\lambda y.zy) \\ &{}^{(\alpha)(\alpha)} &&= [z \mapsto \lambda x.xy](\lambda t.[z \mapsto t]x)\ ([z \mapsto \lambda x.xy](\lambda u.[y \mapsto u]zy)) \\ &{}^{(2)(3,2+1)} &&= [z \mapsto \lambda x.xy]\lambda t.x\ ([z \mapsto \lambda x.xy]\lambda u.zu) \\ &{}^{(4)(4)} &&= \lambda t.[z \mapsto \lambda x.xy]x\ (\lambda u.[z \mapsto \lambda x.xy]zu) \\ &{}^{(2)(3,1+2)} &&= \lambda t.x\ (\lambda u.(\lambda x.xy)u) \end{aligned}$$

- Everyone should be comfortable with such rewritings...

1 The syntactic landscape
2 Computing with syntactic objects
3 Conclusion

- The usual function call can be modeled by

$$\underbrace{(\lambda x.T_1)}_{(1)}\underbrace{T_2}_{(2)} \to \underbrace{[x \mapsto T_2]T_1}_{(3)}$$

  where (1) is the function, (2) the argument and (3) the result
- For example $\mathbf{II} = \lambda x.x\ \lambda x.x \to [x \mapsto \lambda x.x]x = \lambda x.x = \mathbf{I}$
- This rule is called β-reduction (def later)
- It can be applied anywhere within a term
- A location in a term where it can be applied is called a β-redex

- A judgment is a logical assertion, here[2]: $Term \to OtherTerm$
- An inference rule is a set of judgments $J_1, ..., J_n, J$ such that $J_1 \wedge ... \wedge J_n \Rightarrow J$
  - $J_1, ..., J_n$ are the premises, J is the conclusion
  - written $\dfrac{J_1 \quad \cdots \quad J_n}{J}$
  - An axiom is an inference rule with no premise
- A derivation is a tree of such rules where the leaves are axioms

$$\dfrac{\dfrac{}{J_1} \quad \dfrac{}{J_2} \quad \cdots \quad \dfrac{\dfrac{}{J_3} \quad \cdots \quad \dfrac{}{J_4}}{J_5}}{J_6}$$

see http://en.wikipedia.org/wiki/Inference_rule

[2] There exists various other forms of judgment

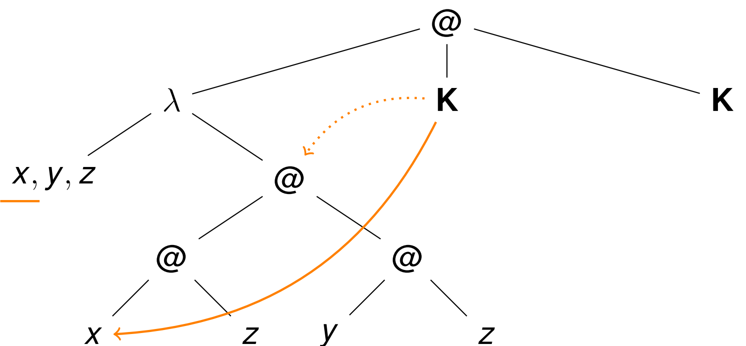
(1) $(\lambda x.T_1)T_2 \to [x \mapsto T_2]T_1$

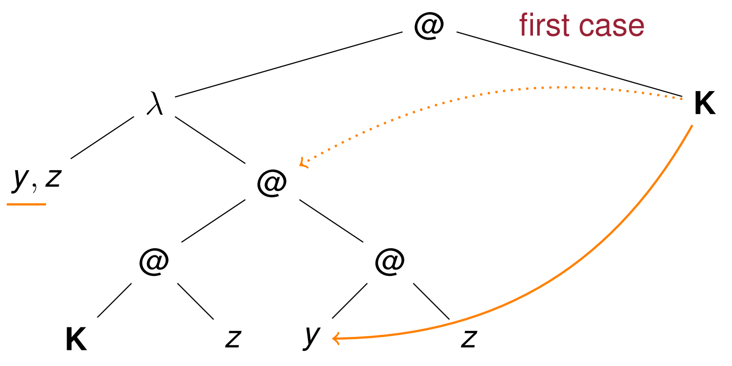
(2) $\dfrac{T \to T'}{\lambda x.T \to \lambda x.T'}$

(3) $\dfrac{T_1 \to T_1'}{T_1 T_2 \to T_1' T_2}$

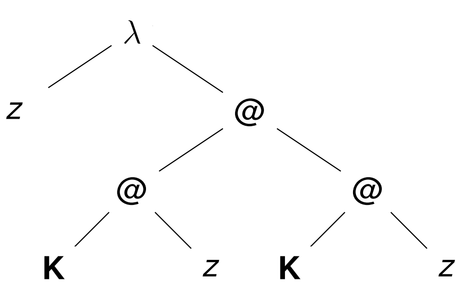
(4) $\dfrac{T_2 \to T_2'}{T_1 T_2 \to T_1 T_2'}$

$\mathbf{SKK} = \lambda x.\lambda y.\lambda z.((xz)(yz))\mathbf{KK}$	def of **S**
$\to \lambda y.\lambda z.((\mathbf{K}z)(yz))\mathbf{K}$	(1)
$\to \lambda z.((\mathbf{K}z)(\mathbf{K}z))$	(1)
$\to \lambda z.(((\lambda x.\lambda y.x)z)(\mathbf{K}z))$	def of **K**
$\to \lambda z.(\lambda y.z(\mathbf{K}z))$	(1)
$\to \lambda z.z$	(1)
$\to \mathbf{I}$	def of **I**



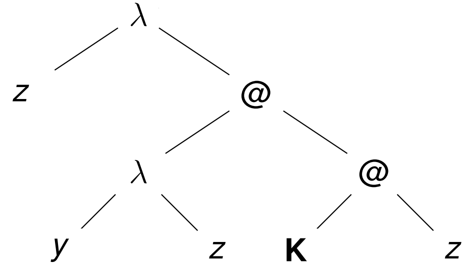


first case



- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$

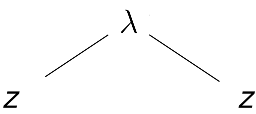
first case



- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$

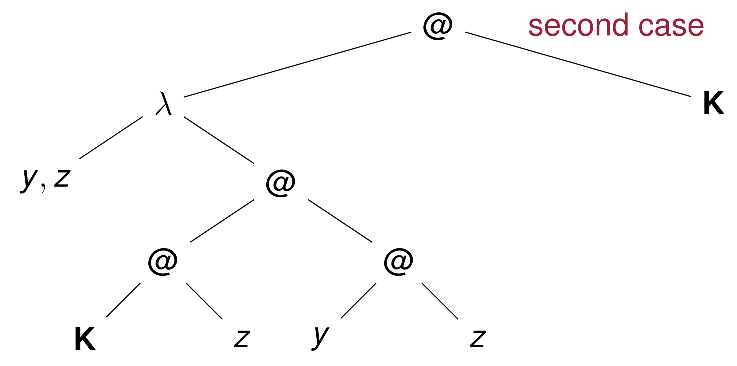
Another look 19 / 26

first case



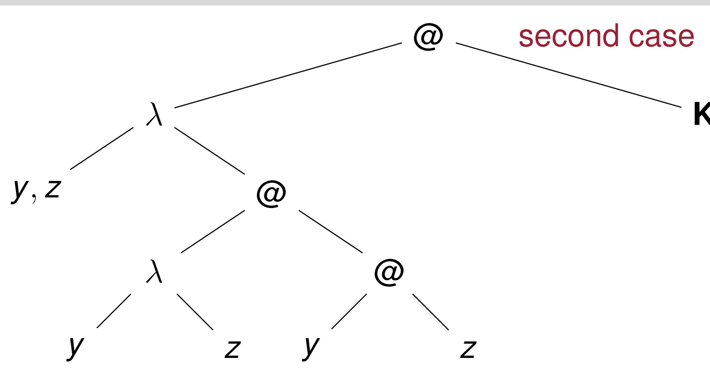
- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$

Another look 19 / 26

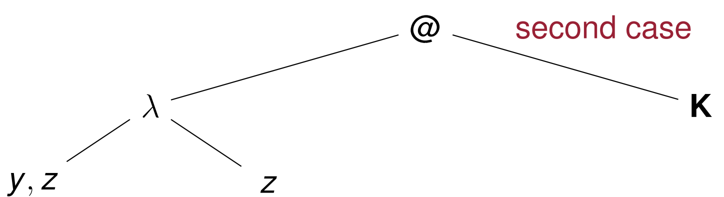


- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$

Another look 19 / 26

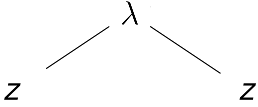


- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$



- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$

second case



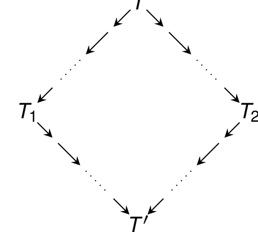
same result!

- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$

- A term T is irreducible or normal, if there exist no term it can reduce to ($T \not\to$)
- If T reduces to T' normal, T' is called a normal form of T
- A reduction sequence is a sequence $T_1 \to \cdots \to T_n$
  - denoted $T_1 \to^n T_n$
  - denoted $T_1 \to^* T_n$ if you don't care about the number of steps
- Often, there is several reduction sequences starting from a term (*e.g.* **SKK**)
- A reduction (resp. a term) is
  - (strongly) normalizing if all (resp. its) reduction sequences are finite
  - weakly normalizing if all terms have (resp. it has) a normal form
- $\Omega = (\lambda x.xx)(\lambda x.xx) \to \Omega$

⚠ β-reduction is not weakly normalizing for $\Lambda_{\mathcal{X}}$

- If T reduces to T_1 and T_2 there exists T' such that T_1 and T_2 both reduce to T'



- It shows that the path of computation is not important
- A term has at most one normal form

Church-Rosser theorem

β-reduction is confluent on $\Lambda_{\mathcal{X}}$

⚠ Some terms reduces indefinitely but has a normal form: $\mathbf{KI}\Omega \to \mathbf{KI}\Omega$ or $\mathbf{KI}\Omega \to^2 \mathbf{I}$

Reduction strategy

- A reduction strategy is a way to choose the β-redex to reduce
- Standard orders
  - Normal order
    - the leftmost outermost reduction
    - always finds the normal form if it exists
  - Applicative order
    - the leftmost innermost reduction
    - only finds the normal form for normalizing terms
  - but both reduce inside functions (rule (2))
- Two other classical strategies (not using rule (2))
  - call by name: resolve application before evaluating the arguments
    - may duplicate computations
  - call by value: evaluate argument before application
    - optimal for sharing of computations

Is $\Lambda_{\mathcal{X}}$ a programming language?

- In theory, yes as everything can be encoded as a λ
  - Turing has proved all computable functions can be written in $\Lambda_{\mathcal{X}}$
- In practice not usable, what is this term[3]?
  - $\lambda xyzu.(x(yzu)u)\lambda xy.(y(yx))\lambda xy.(yx)$
  - 2 + 1
- We extend its core with
  - basic datatypes (integer, boolean, ...)
  - data structures (pairs, lists, ...)
  - recursion
  - ...

  It's the functional core of Ocaml! http://caml.inria.fr/ocaml

[3] We use $\lambda xyz.$ for $\lambda x.\lambda y.\lambda z.$, this notation si called currying

Progress

1. The syntactic landscape
2. Computing with syntactic objects
3. Conclusion

Conclusion

- The λ-calculus
  - anything that is computable can be expressed
  - is often used to study sequential computation
  - close to a programing language (Caml)
  - for the interested [Lal90]
- Used to illustrate fundamental notions
  - variables, scope
  - induction
  - substitution
  - reduction
- Starting point to learn functional programming

N. Bourbaki.
Théorie des ensembles.
Eléments de mathématique. Springer, 2008.

René Lalement.
Logique Réduction Résolution.
ERI Masson, 1990.
The book has been translated in english under the title *Computation as logic* and edited by Prentice-Hall in 1993, ISBN 9780137700097.

PC1 – Let's practice rewriting

Objectives

At the end of the activity, you should be capable of:

- writing and understanding a λ-term;
- compute in the λ-calculus (apply substitutions and reductions).

1 Basic syntax of the λ-calculus

Exercise 1 (*Parenthesis*)

▷ **Remove parenthesis as much as possible for the following λ-terms:**

1.1 $((\lambda y.(\lambda x.((yz)x)))(\lambda x.x))$

1.2 $(\lambda x.((\lambda y.((\lambda x.y)y))x))$

1.3 $((\lambda y.((\lambda z.A)(yy)))(\lambda x.(xx)))$ it is a metaterm, $A \in \Lambda_{\mathcal{X}}$

1.4 $(((ab)(cd))((ef)(gh)))$

Exercise 2 (*Tree representation*)

▷ **Give the tree representation of the following λ-terms:**

2.1 $\lambda x.(\lambda y.yy)zx$

2.2 $(\lambda y.yyy)(\lambda x.xx)$

2.3 $ux(yz)(\lambda v.vy)$

2.4 $(\lambda x.\lambda y.\lambda z.xz(yz))uvw$

2.5 $w(\lambda x.\lambda y.\lambda z.xz(yz))uv$

Exercise 3 (*Free variables*)

▷ **Give the set of free variables of the terms 1.1 to 2.5.**

Exercise 4 (*Substitution*)

▷ **Apply the following substitutions**

4.1 $[x \mapsto \lambda y.xy](\lambda y.x(\lambda x.x))$

4.2 $[y \mapsto \lambda v.vv](\lambda y.x(\lambda x.x))$

4.3 $[x \mapsto \lambda y.vy](y(\lambda v.xv))$

Exercise 5 (*Reduction*)

▷ **Gives all the possible sequences of reductions for the terms 1.1, 2.1, 2.4, then 2.2, 1.2 and 1.3.**

2 Data types in the λ-calculus

Exercise 6 (*Boolean*)

Let **T** and **F** be two combinators defined by $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$.

▷ **Question 6.1:**
Show that the combinator $\mathit{If} = \lambda b.\lambda x.\lambda y.bxy$ **can be the usual if operator.**

▷ **Question 6.2:**
Give the combinators Not**,** And **and** Or

Exercise 7 (*Pairs*)

Pairs are defined by two functions *fst* and *snd*. A pair stores two values, the first being retrieved by *fst* and the second by *snd*. Let's denote a pair containing v_1 and v_2 by (v_1, v_2) then:

$$\begin{cases} \mathit{fst}\,(v_1, v_2) & = v_1 \\ \mathit{snd}\,(v_1, v_2) & = v_2 \end{cases}$$

▷ **Define** *fst* **and** *snd* **if we encode the pair** (v_1, v_2) **as** $\lambda f.f\, v_1\, v_2$**.**

PC2 – Let's practice induction

Objectives

At the end of the activity, you should be capable of:

- defining an object by induction;
- make simple proof by induction.

Exercise 1 (*Subterms*)

▷ **Question 1.1:**
Is there an occurrence of the term $T_1 = yx(xz)$ in the term $T_2 = zyx(xz)x$?

▷ **Question 1.2:**
Is there an occurrence of the term $(\lambda x.\lambda y.\lambda z.(xz)(yz))u$ in the following terms:

$$\begin{cases} T_1 = (\lambda x.\lambda y.\lambda z.xz(yz))uvw \\ T_2 = w(\lambda x.\lambda y.\lambda z.xz(yz))uv \end{cases}$$

▷ **Question 1.3:**
Define the function sub which computes the set of subterms of a term.

Exercise 2 (*Binary Trees*)

A is a set values. A binary tree over A is a either empty or contains a value from A and has a left child and a right child.

▷ **Question 2.1:**
Define inductively the set $\mathcal{B}_A$ of binary trees over A.

▷ **Question 2.2:**
Define the function $|\cdot|$ which compute the number of nodes of a binary tree.

▷ **Question 2.3:**
Define the function h which compute the height of a binary tree. The height is defined the longest path between the root of the tree and a leaf.

▷ **Question 2.4:**
Prove that for any binary tree T, $|T| \leq 2^{h(T)+1} - 1$

Exercise 3 (*Church Integers*)

Let $\underline{0}$ be the combinator defined by $\lambda s.\lambda z.z$. Let it represents zero. Let $\underline{\mathsf{succ}}$ be the function $\lambda v.\lambda s.\lambda z.s\,(vsz)$.

▷ **Question 3.1:**
If we use $\underline{\mathsf{succ}}$ as the usual successor function of integers what is the encoding of $\underline{n}$ for $n \in \mathbb{N}$? Prove your proposition by induction.

▷ **Question 3.2:**
Prove that usual addition, multiplication and exponentiation can be defined by:

$$\begin{cases} \underline{+} = \lambda nm.(n\,\underline{\mathsf{succ}}\,m) \\ \underline{\times} = \lambda nm.(n\,(\underline{+}\,m)\,\underline{0}) \end{cases}$$

▷ **Question 3.3:**
Propose a term for exponentiation.

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Functional programming
Introduction to OCaml

Fabien Dagnat
ELU 610 – C6
1st semester 2019

1. OCaml basics
2. More type constructors
3. Modules
4. Executing and Building
5. Conclusion

- General purpose language developed by INRIA since 1990. . .
- . . . and now widely used by industrials (Airbus, ANSSI, CEA, Be-Sport, Bloomberg, Facebook, Jane Street Capital, Tezos, . . .)
- `http://ocaml.org`
- A book is downloadable at `https://realworldocaml.org`
- This lesson covers only the functional part (chap 1-7)
- Online simple tutorial at `http://try.ocamlpro.com`

`http://ocaml.org/learn/companies.html`

Progress

1. OCaml basics
2. More type constructors
3. Modules
4. Executing and Building
5. Conclusion

λ-calcul ⊂ OCaml

- λ-calcul
  - Variables are strings `beginning_by_lowercase_letter`
  - Application of `a` to `b` is just `a b`
  - (Anonymous) function `function x -> body`
- Some syntactic sugar
  - A function with several arguments `fun x y -> body`
    - equivalent to `function x -> function y -> body`
  - Naming a value `let x = value in body`
    - note that the scope of `x` is explicit (here `body`)
    - equivalent to `(function x -> body) value`
  - Naming a function `let f x y = value in body`
    - equivalent to `let f = fun x y -> value in body`
  - Sequencing `a ; b`
    - equivalent to `let _ = a in b`

Evaluation order

- Computation use call by value
- evaluate arguments before application
- ⚠ evaluation order is undefined between the arguments of a function
- computing `f a1 a2` computes
  1. `f`[1], `a1` and `a2` in a unspecified order
  2. computes the call
- if you need a specific order, use `let`:

```
let f = ... in
let a1 = ... in
let a2 = ... in
f a1 a2
```

[1] the function may be any expression

Toplevel expressions and execution

- A toplevel expression is an expr. not contained in a larger expr.
- A toplevel naming expression (`let`) without a scope (no `in`)
  - has its scope extended to all the following toplevel expr.
  - provides a kind of global naming
- There is two ways of executing OCaml programs
  1. using a(n interactive) REPL[2] (an interpreter), `utop` or `ocaml`
     it reads an expr., evaluates it and then prints the result
  2. using a compiler and then executing the produced executable file
- In a REPL, you enter an expr. and terminates it by `;;`
- The compilers consider that two expr. separated by a (blank) line are toplevel expr. in sequence

[2] Read-eval-print-loop

Practicing

- The identity function
- A function applying a function to a value
- A function composing two functions

Typing

- OCaml is a typed language with
  - primitive types
    - `bool`, `int`, `float`, `char`, `string`, ...
  - type constructors (build new types from existing types)
    - `t list` for lists of type `t`, `t1 * t2` for pairs of type `t1` and `t2`, ...
    - `t1 -> t2` for functions from `t1` to `t2`
- Typing is static: typing correctness is checked before execution
- Types of expressions are inferred (computed) by the compiler
- ⇒ The programmer is not required to give them!
- The REPL prints them for toplevel expr.

```
# let add i = i + 10 ;;
val add : int -> int = <fun>

# add 12 ;;
- : int = 22
```

expressions — types — values

Typing is strict

- Typing is strict, each expression must be correctly typed

```
# add "tutu" ;;
Error:          This expression has type string but an
expression was expected of type int
```

- There is no automatic conversion

```
# add 12.1 ;;
Error: This expression has type float but an expression
was expected of type int
# add (int_of_float 12.1) ;;
- : int = 22
```

- OCaml has no overloading: a name has only one type

```
# let pi = 4.0 * atan 1.0 ;;
Error: This expression has type float but an expression
was expected of type int
# let pi = 4.0 *. atan 1.0 ;;
val pi : float = 3.141592653589793l2
```

Polymorphism

- What is the type of the identity function? `let id x = x`
- `id` can take any value as argument and returns this value
- ⇒ Types may contain type variables `'a`, `'b`, ...
- The type of `id` is ∀`'a.'a -> 'a`
- This is called (universal) polymorphism

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

the universal quantification is implicit

```
# let eval f x = f x ;;
val eval :                      = <fun>

# let compose f g x = f (g x) ;;
val compose :                             = <fun>
```

Primitive data types

- `()` the nothing value of type `unit`
- **`false`** and **`true`** of type `bool`
  - logical operators: **not**, **&&**, **||**, ...
  - comparison operators: =, <>, <, >, <=, >=
- integers of type `int`
  - usual operators: +,-,*, /, **mod**, `int_of_float`, ...
- floating number of type `float`
  - usual operators: `+.`,`-.`,`*.`, `/.`, `**`, `float_of_int`, ...
- `'a'`, `'\n'`, ... of type `char`
- `"\ta string\n"` of type `string`
  - concatenation by `^`
  - conversion of the primitive data types by `string_of_`*type*
  - char at position `i` by `str.[i]`

Type constructors: product

- Tuples (`e1,...,en`) of type `t1 * ... * tn`
  - no function to decompose (see later pattern matching)
  - pairs when $n = 2$, decompose using `fst` and `snd`
- If a vector is represented by a pair, compute its norm

```
# let square x = x *. x ;;
val square : float -> float = <fun>
# let norm c = sqrt (square(fst c) +. square(snd c)) ;;
val norm : float * float -> float = <fun>
# norm (2.0,-1.0) ;;
- : float = 2.23606797749979
```

- A function applying two functions to a pair

```
# let apply f g c = f (fst c), g (snd c) ;;
val apply : ('a -> 'b) -> ('c -> 'd) -> 'a * 'c -> 'b * 'd = <fun>
# apply (fun x -> x + 1) (fun x -> x - 1) (4,4) ;;
- : int * int = (5,3)
```

Patterns

- A pattern is an expression made of
  - value constructors and values
  - variables (only one occurrence for each variable)
  - holes: `_`

  (`1,`**`true`**), 1, (`1,_,x`) are patterns
- A pattern may match a value
  - if they have the same (constructor) structure
  - variables and holes match any value

  (`1,_,x`) matches (`1,"er",'a'`) but neither 1 nor (`2,"er",'a'`)
- When a pattern matches a value, its variables are bound to the corresponding parts of the value
  when (`1,_,x`) matches (`1,"er",'a'`) it binds `x` to `'a'`
- The let syntax is **`let`** *pattern* = *expression* [**`in`** *expression*]

Pattern matching

- Main control structure, used to decompose values
- A pattern matching case is a pattern and an expr.
  - when "applied" to a value, it can succeed or fail
  - if it succeeds, expr. is evaluated with the variables bound
  - syntax: *pattern* -> *expression*
- A pattern matching is a sequence of pattern matching cases
  - when "applied" to a value, it uses the first case to try to match
  - if it fails, the next case is used
  - and so on, until one case matches
  - if none of the cases matches, there is a `Match_failure` exception
  - **`function`** p_1 -> e_1 | ... | p_n -> e_n[3]
  - **`match`** *e* **`with`** p_1 -> e_1 | ... | p_n -> e_n
    - equivalent to (**`function`** p_1 -> e_1 | ... | p_n -> e_n) *e*
- If a case is useless or is missing, the typer will raise a warning

[3] **`fun`** can only have one case

Practice of pattern matching

- Compute $a \Rightarrow b$ for a pair of boolean (a, b)

- A function testing if an integer is zero

Lists

- Lists are built from
  - the empty list: []
  - an element e and a list l: e`::`l
    - e is the head of e`::`l
    - l is the tail of e`::`l
- Lists are monomorphic
  - all elements in a list must have the same type
  - a list of elements of type `t` is of type `t list`
- Syntactic sugar: $[e_1; \ldots; e_n]$ is equivalent to $e_1 :: \ldots :: e_n :: []$
- Utility functions for lists
  - concatenation `@`
  - lots of utility functions in library `List` (head by `hd`, tail by `tl`, ...)

Recursive functions

- In functional languages, iteration is done by recursion
- Recursive functions are defined by **`let rec`**
- Recursion is often combined with pattern matching

```
# let rec insert elt = function
  | [] -> [elt]
  | h::t when elt <= h -> elt::h::t
  | h::t -> h::(insert elt t) ;;
val insert : 'a -> 'a list -> 'a list = <fun>
```

- Computing Fibonacci numbers

Functions are values

- Functions can
  - take functions as arguments
  - return functions
  - can be applied partially
- For example, applying a function on all elements of a list

```
# let rec iter f = function
  | [] -> ()
  | h::t -> f h; iter f t ;;
 val iter : ('a -> 'b) -> 'a list -> unit = <fun>
# iter print_string ["a";"b";"c";"n"] ;;
abcn
- : unit = ()
# let print_list = iter print_string ;;
val print_list : string list -> unit = <fun>
```

Progress

1. OCaml basics
2. More type constructors
3. Modules
4. Executing and Building
5. Conclusion

Another product type constructors: records

- Records: product types naming the sub-elements

```
# type ratio = { num : int ; den : int } ;;
type ratio = { num : int; den : int; }
```

- A value of type `ratio` can be defined by

```
# let r1 = { num = 1 ; den = 16 } ;;
val r1 : ratio = {num = 1; den = 16}
# let r2 = { r1 with num = 3 } ;;
val r2 : ratio = {num = 3; den = 16}
```

  the order in which the fields are given is unimportant

- The field value are accessed by their name

```
# let add r1 r2 = {
  num = r1.num * r2.den + r2.num * r1.den ;
  den = r1.den * r2.den } ;;
val add : ratio -> ratio -> ratio = <fun>
```

Sum types

- Enumerations

```
# type dir = North | South | East | West ;;
type dir = North | South | East | West
```

- Generalized by variants

```
# type number = Int of int | Float of float | Error ;;
type number = Int of int | Float of float | Error
```

- `Int` 8, `Float` 5.4 and `Error` are of type `number`

```
# (Int 8, Float 5.4, Error) ;;
- : number * number * number = (Int 8, Float 5.4, Error)
```

- Values of variant types are manipulated by pattern matching

```
# let print_number = function
  | Int n -> print_int n
  | Float f -> print_float f
  | Error -> print_string "error" ;;
val print_number : number -> unit = <fun>
```

Sum types II

- Sum types can be parameterized

```
# type 'a option = None | Some of 'a ;;
type 'a option = None | Some of 'a
```

- Sum types can be recursive

```
# type 'a list = [] | :: of 'a * 'a list ;;
type 'a list = [] | :: of 'a * 'a list
```

- Both the option and list types are already defined in OCaml
- ⚠ In fact in OCaml, the variant constructors must be
  1. a capitalized identifier
  2. []
  3. `::`[4], it will be treated as a binary infix constructor

[4] to be correct the type declaration should surrounds it by parenthesis

1 OCaml basics

2 More type constructors

3 Modules

4 Executing and Building

5 Conclusion

- A module is a set of type, value and function definitions[5]
- A module has
  - a signature defining its public interface
    - type definitions and type declarations for values and functions
  - a structure defining its content
    - any OCaml code
- Elements of the signature must be part of the structure
- In another module, one can use a **public** element `elt` of a module `M`
  - by `M.elt`
  - by `elt` if the module was previously opened by **open** `M`
- A `filename.ml` file is a module `Filename`
  - if there is a `filename.mli`, it provides its signature
  - else everything is public (which is a bad practice)

[5] and modules but we won't cover that

- One of the interest of modules is abstracting (hiding) types
- The following signature abstracts `ratio`

```
type ratio
val create : int -> int -> ratio          A constructor
val add : ratio -> ratio -> ratio         A manipulator
val print : ratio -> unit                 A destructor
```

- Code outside the defining module of an abstract type
  - cannot use its implementation
  - can only manipulate value through the offered functions

⇒ Changing the implementation of `ratio` does not impact clients

- Abstracting internal functions and values is also a good idea
- The signature is the ApplicationProgrammingInterface
  - it generally includes constructors, manipulators and destructors for the abstract types

- Exceptions are declared by **exception**

```
# exception Empty_list of string ;;
exception Empty_list of string
```

- Raised by `raise`

```
# let head = function
  | [] -> raise (Empty_list "bouh!")
  | hd :: tl -> hd ;;
val head : 'a list -> 'a = <fun>
```

- Caught by **try with**

```
try
  head []
with
  | Empty_list msg -> print_endline msg ;;
bouh!
- : unit = ()
```

1 OCaml basics

2 More type constructors

3 Modules

4 Executing and Building

5 Conclusion

- `ocaml` REPL (we use `utop`)
  - compiles and executes immediately
  - prints value and types
  - provides a simple line editor (bash default binding)
  - `#use "toto.ml";;` loads and execute every expr. of `toto.ml`
- Two compilers exist
  - a bytecode compiler `ocamlc` (with bytecode interpreter `ocamlrun`[6])
  - a native compiler `ocamlopt` that directly produces executable files
- Both are three steps compilers (`XX` means `c` or `opt`)
  - compile signatures by `ocamlXX -c YY.mli` to produce `YY.cmi`
  - compile modules by `ocamlXX -c YY.ml` to produce `YY.cmZZ`
    - `ZZ` = `o` if `XX` = `c` and `x` if `XX` = `opt`
  - linking of all the need modules by `ocamlc -o WW YY1.cmZZ YY2.cmZZ` to produce the executable `WW`
- For a module without signature all its content is put in the `cmi`

[6] Invoking `ocamlrun` is optional since the bytecode file already invoke it

- All files are compiled in the directory `_build`
- It groans if it finds compilation artefacts elsewhere!
- It has a target `X.byte` or `X.native` to indicate the compiler
  - `ocamlbuild -libs unix main.native`
  - `X` become the name of the executable
- Finds all the dependencies (hence the compilation order) alone
- Can run if you add `--` followed by the command line args.
  `ocamlbuild main.byte -- file.txt`
- Configuration file `_tags` for a finer control of build
- `https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc`

1 OCaml basics

2 More type constructors

3 Modules

4 Executing and Building

5 Conclusion

- Short introduction to the functional core of OCaml
  « Computing values not modifying variables »
- It is our objective during this module
- We ignore
  - imperative features
  - objects
  - first class modules, ...
- You need to practice...
- `http://ocaml.org`
- `https://realworldocaml.org` (chap 1-7 and 16)

TP4-6 – Discovering OCaml

Objectives

At the end of the activity, you should be capable of:

- build simple functional OCaml modules;
- run OCaml programs from the interpreter;
- build and run OCaml programs from the command line;
- discover the rest of the language by yourselves.

Introduction

OCaml is a functional programming language. In other words, it is a programming language where functions are *first class values*. Functions can be manipulated like any other kind of value: passed as parameters to other functions, returned as result of a call to a function, stored in a data structure, ...

OCaml is a language:

- statically typed by inference: the use of values and variables is verified at the time of compilation; information of type does not need to be provided by the programmer, it is computed by the compiler.
- offering parametric polymorphism: if a function does not explore the whole structure of one of its arguments, it has a type not entirely determined (a variable type).
- whose allocations and deallocations of data in memory are automatically managed by a garbage collector.
- including imperative features: it is possible to use imperative control structures and to physically manipulate some values (arrays, references, ...). Imperative feature tend to use side effects.
- providing a comprehensive mechanism of exceptions.
- which has an interpreter (`ocaml`), a virtual machine (`ocamlrun`), two compilers (one for the virtual machine `ocamlc` and a native `ocamlopt` one) and development tools (execution tracing, dependencies management, performance analysis, package deployment, testing, ...).

An OCaml program can be structured using two approaches: by modules or by classes in an object oriented fashion. The choice between these two models of structuration offers a great flexibility to the language. They are dual the modules facilitating the extension of the treatments and the objects facilitating data extension. As part of the course, we will not explore the object aspects of language. The reader interested by this aspect is referred to the referenced documents presented below.

In ocaml, a module is a compilation unit that groups together data and code that are described by an interface. The language integrates multiple notions that allow advanced manipulation of modules (parametric modules, functors, ...). When we will practice, we will limit ourselves to simple modules *e.g.* separate files. This document will describe in more details the way to build a module in the section B.2.

The associated course mainly focused on functional concepts which you probably don't know. Our objective, here, is to practice this functional part of OCaml. In annex A, you will find a partially redacted version of what has been seen during the course. To discover the objects and imperative parts of OCaml, you can consult:

- The official manual `http://caml.inria.fr/pub/docs/manual-ocaml`
- The OCaml portal `http://ocaml.org`
- A page of this portal containing numerous links to books on OCaml: `http://ocaml.org/learn/books.html`. Among these books, I would advise mostly to read:
  - `https://realworldocaml.org`
  - A french teaching book: `http://caml.inria.fr/pub/docs/oreilly-book/index.html`

We will also use `opam` the OCaml package manager (`https://opam.ocaml.org`). This tool allows downloading, compiling and installing libraries for OCaml. It installs software in the directory `.opam`[1] in your root directory and adds access to these libraries to your environment[2].

The various basic tools and `opam` are installed in the lab classes.

1 An introduction

In this section, we recommend using the interpreter (sometimes referred to as an interaction loop).

1.1 The interpreter

The ocaml interpreter is started by the command `ocaml`. We are going to use a more user-friendly version of this interpreter: `utop`[3]. Therefore, we need to install it `opam install utop`.

The figure 1 contains a screenshot of `utop` when launching it. The *prompt* of the interpreter is the character `#`. Once this character is displayed, the interpreter reads the data input until meeting `;;`. The sentences can be expressions (which give a result) or definitions that introduce new variables, types, ... (*e.g.* `let` statement). The interpreter reacts to an expression by displaying the result of its evaluation preceded by its type and to a definition by giving the type of the defined value and possibly its value. For example :

```
# 1+2*3 ;;
- : int = 7

# let pi = 4.0 *. atan 1.0 ;;
val pi : float = 3.14159265358979312

# let square x = x *. x ;;
val square : float -> float = <fun>
```

---

[1] Before the first use of `opam`, `opam init`, which is responsible for creating the directory `.opam`.

[2] The command `` eval `opam config env` `` is responsible for this initialization. If you want to run it for each new terminal, put it in your `.profile.perso`.

[3] `https://github.com/diml/utop`

Figure 1: `utop` interpreter

```
# square(sin pi) +. square(cos pi) ;;
- : float = 1
```

Directives allow you to interact with the interaction loop. They are distinguished from OCaml expressions because they begin with a sharp `#`. For example, to leave the loop use the `#quit` directive (followed by `;;`). Many directives are available, some specific to `utop` other inherited from the standard interpreter. Among those we will use:

- `#use` which loads and interprets the contents of a file,
- `#directory` to add a file search directory,
- `#cd` to change the current directory of the interpreter,
- `#load` to load a compiled file,
- `#trace` / `untrace` to obtain / suspend traces of calls to the target function,
- ... see `http://caml.inria.fr/pub/docs/manual-ocaml/toplevel.html`.

A line management mechanism is offered by `utop` and follows bindings of `bash` [4], you can modify and move forward / backward through the history (the arrows to go up and go down). Finally, `utop` offers a mechanism for completion, its lower bar dynamically displays the current possible completions, complete by using its first proposal using the tab key.

1.2 Functions

Exercise 1

The `map` function takes a function `f` and a list `l`. It must return the list of the results of applying `f` to elements of `l`.

▷ **Question 1.1:**
Without using the interpreter give the type of `map`.

[4] See `# utop_bindings` for the shortcuts of `utop`.

▷ **Question 1.2:**
Propose an implementation of `map` and check its type.

Exercise 2

Same exercise with the function `iterate` which takes an integer n and a function f and returns f^n[5].

2 More OCaml

Exercise 3 (*Binary trees*)

▷ **Question 3.1:**
Propose a type for binary trees.

▷ **Question 3.2:**
Use this type to realize a binary search tree for integers. Such a tree has the following invariant: for every node, the values contained in all the left sub-tree are smaller to the one of the node and the values contained in all the right sub-tree are larger.

For this define a function `add` that add an integer in a binary search tree.

▷ **Question 3.3:**
Use the previously defined function to implement a function sorting list of integers.

Exercise 4 (*Card deck*)

▷ **Question 4.1:**
Propose a type to represent the cards of an usual card game.

▷ **Question 4.2:**
Define a function `all_the_cards` which takes a color as parameter and builds a list containing all the cards of the given color.

▷ **Question 4.3:**
Define a function `string_of_cards` which converts a card to a string representing its value and color.

3 A functional data structure

Exercise 5 (*List with position*)

The idea of this exercise is to implement (efficiently) a notion of list with a position. It is a stateful data structure that maintain a list of elements together with a current position in this list of elements.

So for example, if we suppose that the elements of the list are integers. Here is such a list 117 34 [55] 3 where the position is indicated by the blue box.

[5] means $f(f(\ldots f()))$

A naive implementation could be done by a pair composed of the list of elements and an integer storing the position. It would not be efficient to access the current element as we would have to find it every time. In a language with pointers (such as Java or C), such a list would be a pair formed of the list of elements and a pointer to the current element. The complexity of accessing the current element would then be in $O(1)$. In a functional language, we do not have pointers[6].

The trick here is to see that such a list is a triple when not empty: the list of the elements before, the current element and the list of the elements after. Using our exemple, we have `[117;34],55,[3]`. The accessing current element is trivial.

▷ **Question 5.1:**
Propose a type, a constructor function to create such lists and a `current` function that returns the current element. The list must polymorphic. The constructor function could, for example, take a list of elements and initialize the position to the first element of this list.

In order to be able to test your implementation do not hesitate to define other useful functions.

▷ **Question 5.2:**
What is the cost of moving the position one element right? What is the cost of moving the position one element left? Could we do better?

Define efficient `move_left`, `move_right`, `add_left` and `add_right` functions (it may require to modify the code already done).

▷ **Question 5.3:**
Define a iter and a map function for your list.

4 Formal calculus

This section contain a longer problem. The objective is to build a simple calculator of arithmetic expressions. un évaluateur

Exercise 6 (*Formal calculus*)

▷ **Question 6.1:**
Propose a type to describe simple expressions containing floating point numbers and the four basic operators ($+$, $-$, $*$, $/$).

▷ **Question 6.2:**
Define a function `eval` to evaluate an expression.

▷ **Question 6.3:**
Extends the type of expression to make it possible to use variables. A variable is a string that appears in an expression. During evaluation, the variables will be given a value.

▷ **Question 6.4:**
Modify the fonction `eval` to receive an environment. An environment is an association list associating values to variables. Meeting a variables not defined in the environment should lead to an error.

[6] They exist in OCaml but we will stick to a pure functional style.

▷ **Question 6.5:**
Write a function `string_of_expr`.

▷ **Question 6.6:**
Define a function `derive` qui that derive an expression with respect to one variable given as argument.

▷ **Question 6.7:**
Propose a function `simplify` that simplifies an expression using the following rules:

$$\begin{cases} -0 = 0 \\ \forall e \quad e + 0 = 0 + e = e \\ \forall e \quad e \times 0 = 0 \times e = 0 \\ \forall e \quad e \times 1 = 1 \times e = e \end{cases}$$

A Basics

OCaml comments are contained between `(* ... *)`. Comments may be nested inside other comments.

A.1 Types

In OCaml, the primitive types are:

- `unit` which contain a unique value `()`,
- **integers** (`int`) with their usual operations (`+`,`-`,`*`, `/`, `mod`, `int_of_float`, ...),
- **floating point numbers** (`float`) with their usual operations (`+.`,`-.`,`*.`, `/.`, `**`, `float_of_int`, ...),
- **booleans** `false` and `true` (`bool`) with classical logical operators and usual comparison operators (`=`, `<>`, `<`, `>`, `<=`, `>=`, `not`, `&&`, `||`),
- **characters** (`char`) between `'` with the usual special characters (`\t`, `\n`, ...),
- **strings** (`string`) between `"` with concatenation `^` ; all previous types may be converted to strings by function of the form `string_of_`*type* ; we can get the character at position i of a string by `tab.[i]`[7],
- **tuples** (`_ * _ * _`) the separator character is `,`, pairs have `fst` and `snd` operators ; larger tuples must be destructured using pattern matching,
- **lists** (`_ list`) between `[]` with the separator character `;`, the list constructor is `::` that add an element to the head of a list, there exists also an operator for concatenation `@`.

For more information, please consult the OCaml manual and more precisely the part on the library (part IV) of `https://caml.inria.fr/pub/docs/manual-ocaml-4.05/`.

[7] Note that OCaml is changing to an immutable string type, you may encounter code modifying strings but we are not going to do this. If you need mutable strings use the type `bytes`.

A.2 Control structure

Usual imperative control structure exists in OCaml: choice `if then else`, sequence `;`, blocks `begin ... end`, iterations `for i = e1 to e2 do e3 done` and `while e1 do e2 done`.

The main control structure is pattern matching. A pattern is a partially constructed value containing *holes* (in fact variables not yet defined). Matching is then an operation making it possible to compare a pattern with a value, if the two entities match (*i.e.* have the same form), the holes (the free variables) are filled by the corresponding sub-values (they are defined). The process is comparable to regular expressions.

The matching may fail, in which case the following case is used or an exception is raised if no other case is available (see example 2, below). Notice that the interpreter (and compiler) emits a warning if a pattern matching is incomplete and may therefore fail.

For example:

```
# let a = ([2;3;4;5],1) ;;
val a : int list * int = ([2; 3; 4; 5], 1)

# match a with (c,1) -> c ;;
Warning: this pattern-matching is not exhaustive. Here is an example of a
value that is not matched: (_, 0)
- : int list = [2; 3; 4; 5]

# match a with (c,2) -> c ;;
Warning: this pattern-matching is not exhaustive. Here is an example of a
value that is not matched: (_, 0)
Exception: Match_failure ("", 1, 0).

# match a with (2::c,1) -> c ;;
Warning: this pattern-matching is not exhaustive. Here is an example of a
value that is not matched: ([], _)
- : int list = [3; 4; 5]
```

A (free) variable can only occur once in the pattern. There exists a "hole" pattern `_` match everything but not binding the resulting data to a variable.

Pattern matching can be done by `match` or `function`. It is also the basics structure for definition functions. Lastly, the handling blocks for exceptions use also pattern matching. For example, an insertion sort can be implemented like follows.

First, there is a function to insert a value in a sorted list:

```
let rec insert elt lst =
 match lst with
   [] -> [elt]
 | h::t -> if elt <= h then elt::lst else h::(insert elt t)
```

Then it is used to sort any list:

```
let rec sort lst =
 match lst with
   [] -> []
 | h::t -> insert h (sort t)
```

The value returned by a function is the value of its last expression.

This function also illustrates polymorphism. Indeed, as it does not use the structure of the elements, it is independant of it and has the following type: `'a list -> 'a list` where `'a` is a type variable.

As in all the functional languages, the approach to iterate is to use recursion. Definitions use `let rec`. During this courses, we forget about imperative control structure and focus on functional constructs. It will be a constant requirements for all the codes you produce.

Notice also that generally, in OCaml, data structure are immutable. Once a list has been defined one cannot modify its content. You only can build new lists.

A.3 Functions

In OCaml, a function is a first class value. It can be given as argument to another function. For example, it is possible to define a function `iter` that wait a function `f` and a list `l` and applies sequentially `f` to all elements of `l`.

```
let rec iter f l =
 match l with
 | [] -> ()
 | h::t -> f h; iter f t
```

This function has type `('a -> 'b) -> 'a list -> unit`. It can be used to prints the elements of a list of strings, the printing function being `print_string`:

```
# iter print_string ["a";"b";"c";"n"] ;;
abc
- : unit = ()
```

In OCaml, one can partially apply functions. It consists in providing less argument that required during the call. The result is then another function expecting the remaining arguments:

```
# let print_list = iter print_string ;;
val print_list : string list -> unit = <fun>

# print_list ["a";"b";"c";"n"] ;;
abc
- : unit = ()
```

B More OCaml

B.1 Compilation

OCaml programs can also be compiled. In a file, all related definitions and expressions are collected to define a module (the `;;` is not required anymore). A signature can also be defined in a `mli` file selecting which declarations are exported. This file is then compiled using the command `ocamlc` (or `ocamlopt`). This operation needs two steps:

1. compilation of all required modules `ocamlc -c ...`
   - compile first the signature if there one; it produces a `cmi`
   - then compile the module; it produces a `cmi` if there is no signature (`mli` file)
2. linking all the obtain compiled artefacts to produce an executable `ocamlc -o prog ...`

The compiled files (or modules) use file extension `cmo` (or `cmx`). The result of linking is a *bytecode* file executable by the virtual machine (`ocamlrun`). By default, this file begins by `#!/usr/bin/ocamlrun`[8] which makes it executable on most systems without explicitly launching `ocamlrun`.

The distribution of OCaml contains an automatic building tool `ocamlbuild`. A run of ocamlbuild has a target provided by the user. It analyzes this target and its dependencies to find all compilation operations required. It then compiles all the needed modules in the right order in the sub-directory `_build` of the current directory. It produces an error if it finds compilation artefacts outside the `_build` directory! They must be removed. There exists two kind of targets depending on the compiler one wants to use `ocamlc` (`.byte`) or `ocamlopt` (`.native`). For example:

```
ocamlbuild -libs unix main.native
```

compile the file `main.ml` and all its dependencies with `ocamlopt`. It will also link the program with the `unix` library and will produce an executable named `main.native`. Lastly, it will create a symbolic link in the current directory to the produce executable.

The tool `ocamlbuild` can also run the built executable if one add `--` followed by the command line arguments.

```
ocamlbuild main.byte -- file.txt
```

launch all the compilation and then run the program `main.byte` passing it the parameter `file.txt`.

A configuration file enable a finer grain control on the builds (file `_tags`) and ocamlbuild includes a *plugin* mechanism to support extensions (made in OCaml). The interested reader is invited to read the documentation of the tool at `https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc`.

B.2 Modules

A module is a set of definitions of types, values, functions, exceptions, ...). It has an *signature* and a *structure*. The signature defines the (external) interface of the module. Each of these public entities can be reused by other modules. The structure must define the entities declared in the signature (with compatible structure). The entities of the structure that are not declared in the signature can not be used by other modules. This makes it possible to abstract types (*i.e.* not manipulable see section B.4).

In its simplest embodiment, a module is a file containing the declarations of the module (extension `ml`). Its signature shall then be in a file with the same name and extension `mli`. In this case, the name of the module corresponds to the name of the file capitalized.

When compiling, the signature is compiled into a file of extension `cmi`. If no signature is provided, all entities of the module (*i.e.* the file) are available (a `cmi` file is automatically generated).

In a program, the use of a value `toto` defined in another module `Tutu` must be prefixed by the name of the module: `Tutu.toto`.

In fact, the modules are much more powerful than presented above, you can consult `http://caml.inria.fr/pub/docs/manual-ocaml/moduleexamples.html` to read more.

B.3 Constructed types

In OCaml, it is possible to define new types (syntax `type truc =`). Theses types can be parameterized (syntax `type 'var truc =`). These new types are often aliases of already existing types. All values of the model types are also of the alias type. Beware that if the alias is then abstracted the two types (the model and the alias) become incompatibles. For example, if you declare `type id = int` and then hide the realization of `id` then `id` (which are integers) and `int` (which are also integers) won't be compatible.

One of the type constructor is the sum constructor which allow to define **variants**. Such a type is the result of a definition of the following form:

```
type name =
 | Name1 of t1
 | Name2 of t2
 | ...
 | Namek of tk
```

This type contains all the values built with the (value) **constructors** `Name1` to `Namek`[9]. Building a new value of this type is done by `Name1(toto)` for example if `toto` is a value of type `t1`.

These types are then manipulated using pattern matching.

A sum type can be recursive when one of its sub-element as a type using the sum type. The type `list` is an example of such a recursive sum type, it is defined by:

```
type 'a list =
 | []
 | :: of 'a * 'a list
```

Another sum types predefined in OCaml is the type `option`:

```
type 'a option =
 | None
 | Some of 'a
```

OCaml also has support to define records (named product types):

```
type ratio = { num: int; denum: int }

let add r1 r2 = { num = r1.num * r2.denum + r2.num * r1.denum; denum = r1.denum * r2.denum }

add {num=1; denum=3} {num=2; denum=5}
```

B.4 Type abstraction

One of the advantages of the concept of modules is the possibility of abstracting a type. Indeed, it is possible to declare a new type (in a signature) without showing (and making accessible) its realization. By example, the type `ratio` can be abstracted by the signature (file `MyRatio.mli`):

```
type ratio
val add : ratio -> ratio -> ratio
val num : ratio -> int
val denum : ratio -> int
val create : int -> int -> ratio
val print : ratio -> unit
```

The realization of the module can the be in the file `MyRatio.ml`:

[8] the path may vary!

[9] Beware capitalization!

```
type ratio = { num: int; denum: int }

let add r1 r2 = { num = r1.num * r2.denum + r2.num * r1.denum; denum = r1.denum * r2.denum }

let num r = r.num
let denum r = r.denum

let create n d = { num = n; denum = d }

open Printf
let print r =
 printf "%i/%i" r.num r.denum
```

The implementation of the type `ratio` is no more usable by other modules, the following:

```
let r = Ratio.create 1 1 in
 print_int r.num
```

generate the following compile error:

```
File "UseRatioError.ml", line 2, characters 14-17:
Error: Unbound record field num
```

To use such an abstracted type, the external code must use the functions provided by the module (the API):

```
let r1 = MyRatio.create 1 1
and r2 = MyRatio.create 1 2 in
  MyRatio.print (MyRatio.add r1 r2);
  print_newline ()
```


B.5 Exceptions

OCaml supports exceptions. An exception must be declared by the keyword `exception`, they can be raised by `raise` and they are caught by the construction `try ... with`.

For example, the function `head` below that returns the head of a list may raise an exception when it is given an empty list:

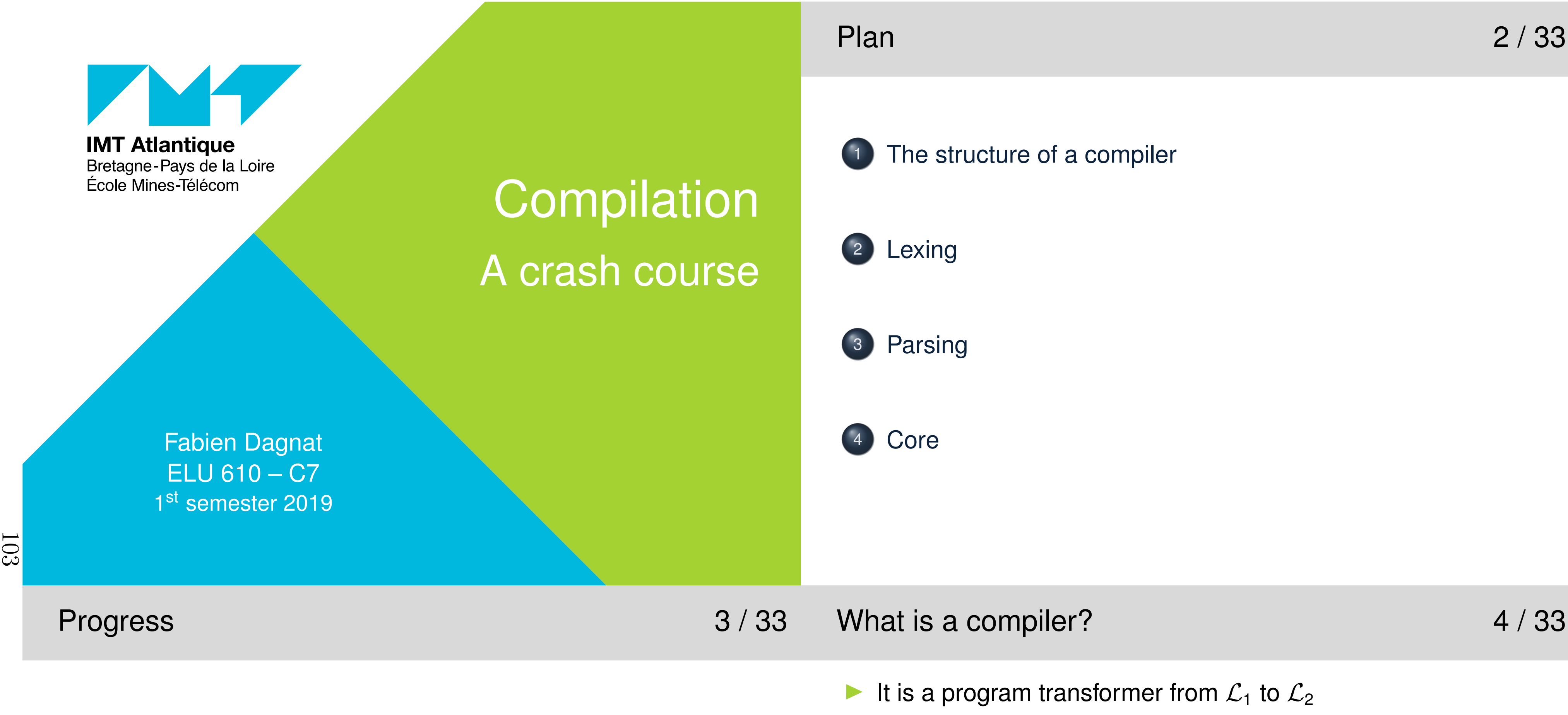
```
exception Empty_list

let head l =
 match l with
 | [] -> raise Empty_list
 | hd :: tl -> hd
```

In the standard library of OCaml, functions are defined to manipulate a notion of dictionary (called association list). The `List` module contains a function `assoc` that takes a key and association list and returns the value associated with the key in the association list. If the key is not present in the association list an exception `Not_found` is raised. Thus, writing a function `name_of_digit` that converts a digit (not a number) to string of characters can be written like follows:

```
let name_of_digit digit =
 try
  List.assoc digit [0, "zero"; 1, "one"; 2, "two"; 3, "three"; 4, "four";
             5, "five"; 6, "six"; 7, "seven"; 8, "eight"; 9, "nine"]
 with Not_found ->
  "not a digit"
```

The `with` part contain a pattern matching and the exception can also contain data.



Progress

1. The structure of a compiler
2. Lexing
3. Parsing
4. Core

What is a compiler?

- It is a program transformer from $\mathcal{L}_1$ to $\mathcal{L}_2$

Source program in language $\mathcal{L}^I$ → **Compiler** → Translated program in language $\mathcal{L}^O$

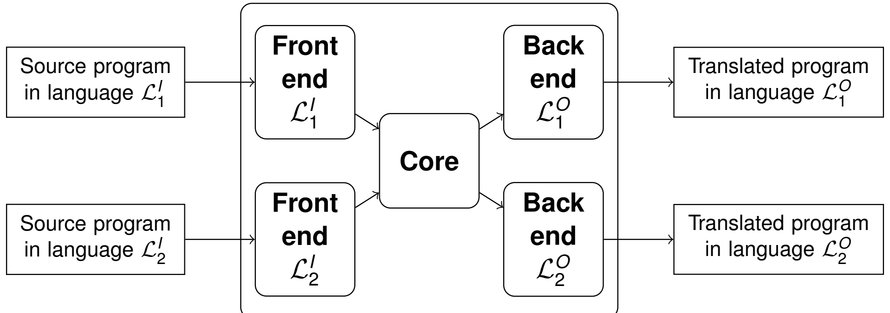
- $\mathcal{L}^O$ is "more" executable than $\mathcal{L}^I$
- Most of the time, a program in $\mathcal{L}_2$ is directly executable
  - either by a machine (*e.g.* $\mathcal{L}^O$ = X64)
  - or by an abstract machine[1] (*e.g.* $\mathcal{L}^O$ = OCaml bytecode)
- Often a compiler is composed of a flow of compilers

$$\mathcal{L}_1 \xrightarrow{\text{compiler}_1} \mathcal{L}_2 \xrightarrow{\text{compiler}_2} \cdots \xrightarrow{\text{compiler}_n} \mathcal{L}_{n+1}$$

[1] an abstract machine is a piece of software acting as a machine

Structure of a compiler

- It is a composed of three stages
  - *front end* in charge of recognizing $\mathcal{L}^I$ (*e.g.* gcc has C, C++, Go, ...)
  - *core* doing the hard work
  - *back end* in charge of emitting $\mathcal{L}^O$ (*e.g.* gcc has X64, ARM, ...)



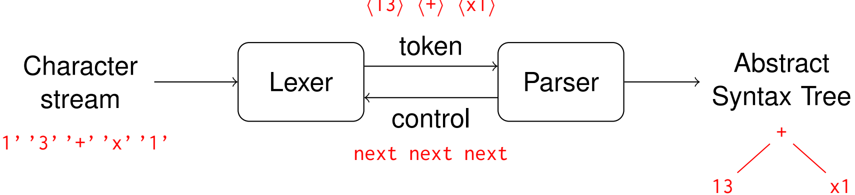
- Several paths are possible

Front end

- It checks whether the program is syntactically correct
  - It belongs to the language $\mathcal{L}^I$
- It must build an internal representation of the program
  - It is an internal data structure of the compiler

⇒ It is highly dependent of the input language

- It is decomposed in two parts
  - lexer recognizes tokens in a character stream
  - parser recognizes sentences in a token stream



Core

- Works on internal data structures
- Is in charge of the verification of validity (typing) of the program
- In charge of the main transformation work, for instance
  - simplify programs by removing useless elements
  - transform function calls
  - transform object oriented access
- Relatively usual software
- Functional paradigm is very adapted for this kind of code
  - recursive functions for the visiting part
  - sum types for representing the various elements
  - product types to add information to the various elements
  - pattern matching for the recognition of structure
  - ...

Back end

- Translates internal data structures in instructions of the target language

```
...
movq $1, %rax
; some code to get the value of x1
addq $26, %rax
...
```

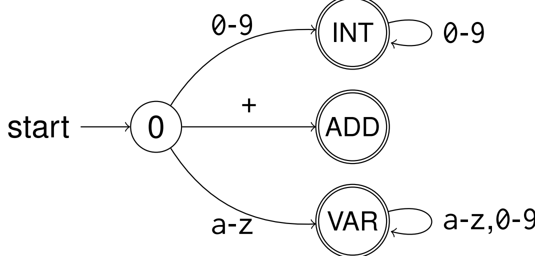
- Implements all optimizations specific to the target
- Complex requiring to master the target machine
- Not in the scope of this introduction...

Progress

1. The structure of a compiler
2. Lexing
3. Parsing
4. Core

Lexical analyzer (*a.k.a* lexer)

- A lexer is in charge of reading enough characters from an entry stream to produce a token
- To be efficient it is generally built as an automaton where
  - transitions correspond to the received characters
  - final state corresponds to the production of the token



- When reaching a final state, it produces a token (data structure for the parser)

Building a lexer

- One way to define such an automaton is to define a set of translation rules
- A translation rule is composed of a pattern and an action
  - the pattern is defined using a regular expression specifying the accepted input
  - the action defines what to do in case of accepting (often just returning the right token)
- For the previous slide example

```
| ['0'-'9']+ { INT((* input converted in int *)) }
| '+' { ADD }
| ['a'-'z']['a'-'z''0'-'9']* { VAR((* input *)) }
```

- A Domain Specific Language: OCamllex
  - a compiler `ocamllex` producing OCaml code for the automaton

OCamllex syntax

- File with extension `.mll`

```
{ (* OCaml code: optional prelude *) }
(* useful regular expressions only for regexp part *)
let ident = regexp
let ident = regexp
(* a group of rules *)
rule ident [ident1 ... identn] = parse
| regexp { (* OCaml code *) }
| regexp { (* OCaml code *) }
(* another  group of rules *)
and ident [ident1 ... identn] = parse
...
{ (* OCaml code: optional postlude *) }
```

Annotations:
- prelude: as is in the result
- regexp: compiled as an automaton
- OCaml code in rules: actions executed on accepting
- rule: translated as a function
- postlude: as is in the result

OCamllex regexp syntax by example

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline
- `[^ '\n' '\r']` ⇒ any character except newline
- `"//"[^ '\n' '\r']*`
- Suppose
  ```
  let digit = ['0'-'9']
  let letter = ['a'-'z''A'-'Z']
  let id_char = (letter | digit | '_')
  ```
- `letter id_char* as id`
- integers
- floating point numbers

Generated OCaml code

- Each `rule name a1 ... an` gives a function `name` taking args
  - `a1`, ... `an`, the user arguments
  - a buffer containing the stream of character of type `Lexing.lexbuf`
- This function matches the characters in the buffer to execute the corresponding accepting action when called
  - it selects the regexp giving the longest part matched
  - in case of equality it selects the first defined
- The standard library module `Lexing` also provides
  - two constructors for buffers: `from_channel` and `from_string`
  - `lexeme buf` returning the matched string of `buf`
- Only one automaton is generated even for several entry points
  - the automaton is determinized and minimized
  - its code is finally put between the prelude and postlude

An example and the flow

- File `formulaLexer.mll`

```
{
 type token = EOF | AND | OR | TRUE | FALSE
}
let space = [' ' '\t' '\n']
rule token = parse
  | space+ { token lexbuf }
 | eof    { EOF }
 | "and"   { AND }
 | "or"   { OR }
 | "true"  { TRUE }
 | "false"  { FALSE }
```

- `ocamllex formulaLexer.mll` produces `formulaLexer.ml`
- It then can be compiled using `ocamlc`

⚠ it can contain errors if the `mll` file contained wrong OCaml code

Progress

1. The structure of a compiler
2. Lexing
3. Parsing
4. Core

Which kind of grammar?

- For most reasonable language syntax, regular expressions are not sufficient
- We must use more powerful grammars but keep efficiency of parsing

⇒ We use context-free grammars (CFG)
  - defined only by production $A \rightarrow m$ where $A \in V$ and $m \in (X \cup V)^*$
- In this course, we will focus on LR(1) parsing by using Menhir[2]
- Menhir
  - offers a DSL for defining grammars in `.mly` files
  - has a tool compiling a grammar spec. to OCaml code (`menhir`)
- Menhir follows a flow similar to OCamllex

[2] `http://gallium.inria.fr/~fpottier/menhir`

An example

- File `formulaParser.mly`

```
%token AND OR EOF TRUE FALSE
%token <string> IDENT
%start< string > formula
%%
formula: c=disj EOF                        { c }
disj:
 | c=conj OR d=disj                        { "("^c^" or "^d^")" }
 | c=conj                                  { c }
conj:
 | s=ident_or_const AND c=conj             { "("^s^" and "^c^")" }
 | s=ident_or_const                        { s }
ident_or_const:
 | id=IDENT                                { id }
 | TRUE                                    { "true" }
 | FALSE                                   { "false" }
%%
```

The token type (the two `%token` lines)

An entry point with its return type (`%start< string > formula`)

Grammar — Actions

Some remarks

- The token type is now generated within the parser

⇒ The lexer does not define it anymore but imports the parser
- Each entry point (`%start`) gives a parsing function of type `(Lexing.lexbuf -> token) -> Lexing.lexbuf -> string`

⇒ The lexer must be given to the parser

```
let compile file =
  try
    let input_file = open_in file in
    let result = formula token (Lexing.from_channel input_file) in
    close_in (input_file);
    printf "read %s\n" result
  with Sys_error s ->
    printf "Can't find file '%s'" file
let () = Arg.parse [] compile ""
```


The result of parsing

- In general, the result of parsing is a data structure representing programs called an Abstract Syntax Tree (AST)
- Defined using recursive sum types

```
type t =
  | Var of string
  | Bool of bool
  | And of t * t
  | Or of t * t
```

- Manipulated by recursive functions

```
let rec string_of = function
  | Var s -> s
  | Bool true -> "true"
  | Bool false -> "false"
  | And(f1,f2) -> "("^(string_of f1)^" and "^(string_of f2)^")"
  | Or(f1,f2) -> "("^(string_of f1)^" or "^(string_of f2)^")"
```

The example revisited

```
%{
 open FormulaAst
%}
%token AND OR EOF TRUE FALSE
%token <string> IDENT
%start< FormulaAst.t > formula
%%
formula: c=conj EOF              { c }
conj:
 | d=disj AND c=conj             { And(d,c) }
 | d=disj                        { d }
disj:
 | s=ident_or_const OR d=disj    { Or(s,d) }
 | s=ident_or_const              { s }
ident_or_const:
 | id=IDENT                      { Var id }
 | TRUE                          { Bool true }
 | FALSE                         { Bool false }
%%
```

Prelude

The new complete flow

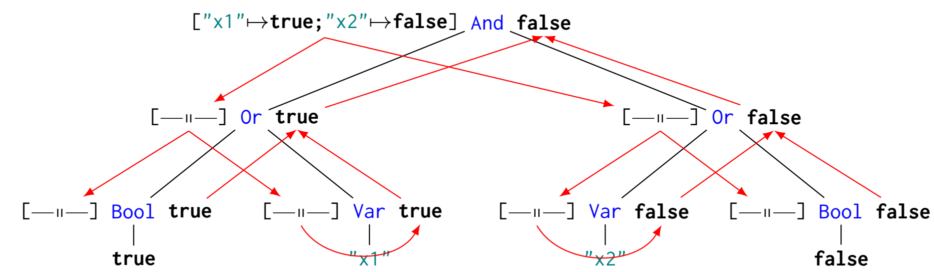
formulaLexer.mll
ocamllex
formulaLexer.ml
Lexer
Character stream
't' 'r' 'u' 'e' ' '
'a' 'n' 'd' ' ' 'x' '1'
formulaParser.mly
menhir
formulaParser.ml
Parser
TRUE AND IDENT("x1")
Lexer
token
control
token token token
formulaAst.ml
Abstract Syntax Tree
And
Bool
Var
true
"x1"

Progress

1. The structure of a compiler
2. Lexing
3. Parsing
4. Core

Evaluation

- One way to execute a program is to use an interpreter
  - often called a Read Eval Print Loop
- Consists in producing a value from an AST
- It uses a recursive visit of the AST to synthesize the value
- While descending into the AST, naming information must be collected



Formalization

- Such visit can be formalized using big step Structural Operational Semantics
  - values are from the boolean algebra $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$ with $\wedge$ and $\vee$
    - $\mathbf{T} \wedge \mathbf{T} = \mathbf{T}, \mathbf{F} \wedge b = b \wedge \mathbf{F} = \mathbf{F}, \mathbf{T} \vee b = b \vee \mathbf{T} = \mathbf{T}, \mathbf{F} \vee \mathbf{F} = \mathbf{F}$
  - an environment function $\mathcal{E}$ mapping variable names to values
    - *dom* gives its domain, $\mathcal{E}(x)$ gives the value associated to x in $\mathcal{E}$
  - judgements of the form $\mathcal{E} \vdash$ AST term $\Rightarrow$ value

$$(1)\ \mathcal{E} \vdash \texttt{Bool}\ \texttt{true} \Rightarrow \mathbf{T} \qquad (2)\ \mathcal{E} \vdash \texttt{Bool}\ \texttt{false} \Rightarrow \mathbf{F}$$

$$(3)\ \frac{x \in dom(\mathcal{E})}{\mathcal{E} \vdash \texttt{Var}\ x \Rightarrow \mathcal{E}(x)} \qquad (4)\ \frac{\mathcal{E} \vdash F_1 \Rightarrow b_1 \qquad \mathcal{E} \vdash F_2 \Rightarrow b_2}{\mathcal{E} \vdash \texttt{And}(F_1, F_2) \Rightarrow b_1 \wedge b_2}$$

$$(5)\ \frac{\mathcal{E} \vdash F_1 \Rightarrow b_1 \qquad \mathcal{E} \vdash F_2 \Rightarrow b_2}{\mathcal{E} \vdash \texttt{Or}(F_1, F_2) \Rightarrow b_1 \vee b_2}$$

Implementation

```
open FormulaAst
let eval env formula =
  let rec eval_rec = function
  | Var s     -> List.assoc s env
  | Bool true  -> true
  | Bool false -> false
  | And(f1,f2) -> (eval_rec f1) && (eval_rec f2)
  | Or(f1,f2)  -> (eval_rec f1) || (eval_rec f2)
  in
  eval_rec formula
```

- Here, as the environment is constant during the visit, it is made global to the visting function (`eval_rec`)
- In implementation, we should take care of errors ($x \notin dom(\mathcal{E})$)

Type

- Evaluation can lead to runtime errors
- Typing "approximates" evaluation to detect a maximum of runtime errors in advance
  - the "value" set is simplified and called type set
  - a new "value" is created to represent errors $\perp$
  - operations are defined on this simplified set
- For our example
  - all booleans values are approximated by the type *bool*
  - $\wedge$ and $\vee$ are both transformed in $\otimes$
    - $bool \otimes bool = bool$ and $\perp \otimes x = x \otimes \perp = \perp$
  - the environment is approximated by a type environment Γ

$$(1)\ \Gamma \vdash \texttt{Bool}\ b : bool \qquad (3)\ \frac{\Gamma \vdash F_1 : b_1 \qquad \Gamma \vdash F_2 : b_2}{\Gamma \vdash \texttt{And}(F_1, F_2) : b_1 \otimes b_2}$$

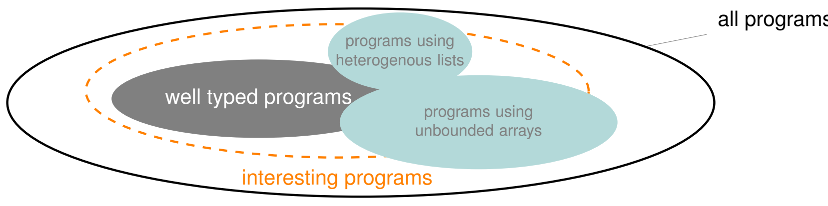
$$(2)\ \Gamma \vdash \texttt{Var}\ x : \Gamma(x) \qquad (4)\ \frac{\Gamma \vdash F_1 : b_1 \qquad \Gamma \vdash F_2 : b_2}{\Gamma \vdash \texttt{Or}(F_1, F_2) : b_1 \otimes b_2}$$

Usefulness of types

Subject reduction theorem (safety)

$$\varnothing \vdash P : \tau \wedge \tau \neq \perp \implies (\varnothing \vdash P \Rightarrow v \wedge \varnothing \vdash v : \tau) \vee \varnothing \vdash P \overset{\infty}{\Rightarrow}$$

- *Well-typed programs cannot "go wrong"* (produce errors)
- Furthermore computing types is much cheaper than evaluating
- ⚠ In general, Halting and Error discovery are undecidable
- Any typing must rejects correct (but too complicated) programs
  - $\Rightarrow$ needs a compromise between flexibility and safety
  - $\Rightarrow$ to achieve safety, runtime checks are often needed



More on types

- Type algebra are often much more complex
  - more values
  - new types during typing...
  - specific relation between types (*e.g.* subtyping)
- Often the developer must add type annotations
  - type checking: given Γ, P and τ, is $\Gamma \vdash P : \tau$ true?
  - the simply typed λ-calculus
    - syntax: $M ::= \ldots \mid \lambda x{:}\tau.M$ and types $\tau ::= \tau \to \tau$
    - typing

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x{:}\tau \vdash M : \tau'}{\Gamma \vdash \lambda x{:}\tau.M : \tau \to \tau'} \qquad \frac{\Gamma \vdash M_1 : \tau \to \tau' \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 M_2 : \tau'}$$

- With no annotation, it is Type Inference
  - typeability: given P finds Γ and τ such that $\Gamma \vdash P : \tau$ is true
  - much harder

Optimization

- The step containing the most difficult algorithms and heuristics
- It consists in transformation of the AST to reduce certain consumption (time, memory, energy, ...)
- Can be generic or specific to a target
- For example
  - for all b, $\mathbf{F} \wedge b = \mathbf{F}$
  - so `And(Bool false,`F`)` can be transformed in `Bool false`
- In real life much more complex!
  - taking out of loop code not depending on the loop
  - loop unrolling
  - propagating constants, inlining small functions
  - removing dead code
  - transforming variables into StaticSingleAssignment
  - tail-call, closure elimination
  - ...

Compilation

- Transform each element of the AST to machine operation
- For example, let's suppose the following machine
  - it manipulates only one bit
  - it has three registers RA, RB and RC (of one bit)
  - it has a memory of 16 bits (M0 to M15) (initialized before running)
  - it supports the following operations
    - set a register to either 0 or 1 SR*xb*
    - loading a form memory to a register LM*i*R*x*
    - the nand[3] NR*x*R*y*R*z* puts R*x* nand R*y* in R*z*
  - during typing, a formula containing more than 16 variables will be rejected and we will build a mapping firm variable names to memory location denoted $\mathcal{M}$
  - translation rules will be of the following form

$$\mathcal{M}, \mathcal{R} \vdash \text{AST term} \leadsto \text{instructions sequence}$$

$\mathcal{R}$ carries the register to hold the result, initialized to RA

[3] it is and followed by not (1 nand 1 = 0 and 0 nand b = b nand 0 = 1)

Example

$$(1)\ \mathcal{M}, \mathcal{R} \vdash \texttt{Bool}\ \textbf{true} \leadsto \mathrm{S}\mathcal{R}1$$

$$(2)\ \mathcal{M}, \mathcal{R} \vdash \texttt{Bool}\ \textbf{false} \leadsto \mathrm{S}\mathcal{R}0$$

$$(3)\ \frac{x \in dom(\mathcal{M})}{\mathcal{M}, \mathcal{R} \vdash \texttt{Var}\ x \leadsto \mathrm{L}\mathcal{M}(x)\mathcal{R}}$$

$$(4)\ \frac{\mathcal{M}, \mathrm{RA} \vdash F_1 \leadsto is_1 \qquad \mathcal{M}, \mathrm{RB} \vdash F_2 \leadsto is_2}{\mathcal{M}, \mathcal{R} \vdash \texttt{And}(F_1, F_2) \leadsto is_1 is_2 \mathrm{NRARBRCNRCRC}\mathcal{R}}$$

$$(5)\ \frac{\mathcal{M}, \mathrm{RA} \vdash F_1 \leadsto is_1 \qquad \mathcal{M}, \mathrm{RB} \vdash F_2 \leadsto is_2}{\mathcal{M}, \mathcal{R} \vdash \texttt{Or}(F_1, F_2) \leadsto is_1 is_2 \mathrm{NRARARANRBRBRBNRARB}\mathcal{R}}$$

- `true or x1 and x2 or false` compiles to SRA1LM0RB NRARARANRBRBRBNRARBRALM1RASRB0NRARARANRBRBRB NRARBRBNRARARCNRCRCRA

- Just a very fast introduction to compilation
- Practice will help concretize!
  - a stack machine language PFX and its execution
  - a micro functional language EXPR, its evaluation and its translation to PFX
- Formalization is important and often forgotten by engineers, that's an error!
- Vocabulary
  - abstract machine, token, sentence, typing, translation rule, pattern, action, abstract syntax tree, interpreter, value, undecidable, type checking, type inference, typeability,
- Acronym
  - LR1, AST, REPL, SOS, SSA

Lecture notes – Compilation with OCaml

Compilation is based first and foremost on the recognition of programs in a stream of characters. The purpose of this section is to discover the practical aspects of *lexical analysis* and *parsing*. Lexical analysis consists in recognizing words of our language in sequences of characters. It is generally followed by parsing that groups these words together to recognize sentences. In the domain, we speak of *token* for words and *syntactic units* for sentences. For the purposes of the following phases of compilation, syntactic units are built in the form of a tree: the so called *Abstract Syntax Tree (AST)*. The figure 1 represents the chaining of these two transformations representing what sometimes is called the front end of the compiler. It aims at building a data structure representing the program in an efficient way. In red are displayed examples of results of each phase. It should be noted that, in general, the lexical analyzer, often called a *lexer*, is driven by the parser that requests tokens whenever it needs them. Hence, the next commands in red.

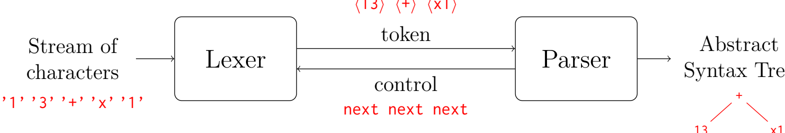


Figure 1: A compiler front end: lexical analysis and parsing.

In general, these analyzers are efficient automata that search for lexical and syntactic patterns using regular expressions. It is difficult and cumbersome to implement an automata. Therefore, *Domain Specific Languages (DSL)* have been proposed to make it easier and smoother.

1 The tools

During this module, we will discover the tools OCamllex whose command is `ocamllex` and Menhir whose command is `menhir`. They are OCaml versions of the standard Unix tools `lex` and `yacc`. The OCaml manual contains a description of `ocamllex` at `http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html` and the Menhir page contains its documentation `http://gallium.inria.fr/~fpottier/menhir/menhir.html.fr`. The chapter 16 of the *Real World OCaml* book can also help you.

OCamllex allows the construction of lexical analyzers as deterministic automatas. It uses a DSL to specify using rules the actions to be executed when recognizing strings. From these rules, it produces an OCaml module. It is, most of the time, used to transform a buffer of characters into a sequence of tokens.

Menhir allows to build easily deterministic automata for parsing. It also provides a DSL dedicated to the specification of rules describing the actions to execute when recognizing sentences. From these rules, it produces an OCaml module. It is, in general, used to transform a sequence of tokens into an AST.

The `menhir` tool produces an LR(1) stack automaton. To use `menhir`, it will need to be installed by `opam install menhir`.

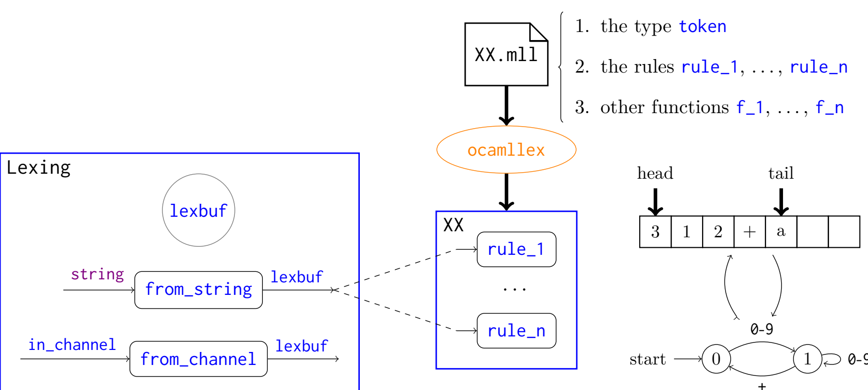


Figure 2: Ocamllex, the big picture.

2 Using Ocamllex

2.1 The syntax of Ocamllex

The files for Ocamllex have the extension `.mll` and follow the following structure:

```
{ (* OCaml code: optional prelude *) }
(* useful regular expressions only for regexp part *)
let ident = regexp
let ident = regexp
(* a group of rules *)
rule ident [ident1 ... identn] = parse
| regexp { (* OCaml code *) }
| regexp { (* OCaml code *) }
(* another  group of rules *)
and ident [ident1 ... identn] = parse
...
{ (* OCaml code: optional postlude *) }
```

The two sections with Ocaml code at the beginning and end of the file are optional. They contain code defining elements (types, functions, etc.) needed for the actions of the rules. The last section can define functions using the functions corresponding to the rules given in the middle section.

The series of statements `let` preceding the definition of rules allows you to name regular expressions. These regular expression can then be reused in the definition of the rules using their names. The regular expressions of Ocamllex follow the syntax presented Figure 3. Here are some examples:

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline
- `[^ '\n' '\r']` ⇒ any character except newline
- `"//"[^ '\n' '\r']*` ⇒ C like line comment

Expression	Meaning
' *char* '	the character *char*
_	any character
eof	end of input
" *string* "	the string *string*
[*ens*]	any character from *ens*, may contain 'c1'-'c2' (all characters between c1 and c2)
[^ *ens*]	any character not in *ens*
regexp *	0 or many time the string matching *regexp*
regexp +	1 or many time the string matching *regexp*
regexp ?	nothing or the string matching *regexp*
regexp1 | *regexp2*	all string matching either *regexp1* or *regexp2*
regexp1 *regexp2*	concatenations of two strings one matching *regexp1* and the other *regexp2*
(*regexp*)	the strings matching *regexp*
ident	the previously defined regular expression *ident*
regexp as *ident*	the result of the matching is bound to the OCaml variable *ident*

Figure 3: Ocamllex regular expression syntax

- Suppose
  ```
  let digit = ['0'-'9']
  let letter = ['a'-'z''A'-'Z']
  let id_char = (letter | digit | '_')
  ```
- `letter id_char* as id` ⇒ identifiers, `id` contains the result
- `digit+ as nb` ⇒ integers
- `digit* '.' digit* (['e' 'E'] ['+' '-']? digit+)? as nb` ⇒ floating point numbers

Each rule, defined by the keyword `rule`, produces a function with the same name (it must be a valid OCaml identifier). If the rule has arguments, the function obtained will have these arguments in addition to one argument (the last) which is a buffer of type `Lexing.lexbuf`. The behavior of this function is to search for a regular expression from its definition list that represents a buffer prefix. The regular expression corresponding to the longest possible prefix is selected [1] and the associated action is executed. The module `Lexing` contains some lexical buffers manipulation functions that the developer can use to define his treatments. This module contains, among other things [2]:

- the type `lexbuf`;
- two constructors for this type: `from_channel` and `from_string` which respectively creates a buffer from an input-output channel or a string;
- the functions:
  - `lexeme`: returns the string recognized by the regular expression. In general, it is easier to use the syntactic construct `as` allows more easily to extract a sub-part of the recognized chain.
  - `lexeme_start`: returns the position index of the beginning of the recognized chain.
  - `lexeme_end`: which returns the position index of end of the recognized chain.

[1] If there are two regular expressions recognizing strings of the same size, the first in the definition order is used.
[2] For more details: `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Lexing.html`.

Each regular expression is compiled by the tool in an automaton. All the automata are merged into a single automaton. This automaton is then determinized and minimized. Its code is inserted between the two portions of OCaml code at the beginning and the end of the file `.mll` to form a `.ml` file (which implements the lexical analysis).

2.2 A first example

To better illustrate Ocamllex syntax, let's look at the file `exprLexer_standalone.mll` that implements a lexer for the expression language EXPR of in exercise 6 from the *Discovering OCaml* document.

```
{
 type token =
  | EOF | PLUS | MINUS | TIMES | DIV | MOD | LPAR | RPAR
  | INT of int | IDENT of string
 let mk_int nb =
  try INT (int_of_string nb)
  with Failure _ -> failwith (Printf.sprintf "Illegal integer '%s': " nb)
}
let newline = (['\n' '\r'] | "\r\n")
let blank = [' ' '\014' '\t' '\012']
let digit = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z']
rule token = parse
 (* newlines *)
 | newline + { token lexbuf }
 (* blanks *)
 | blank + { token lexbuf }
 (* end of file *)
 | eof    { EOF }
 (* integers *)
 | digit+ as nb       { mk_int nb }
 (* commands *)
 | "+"    { PLUS }
 | "-"    { MINUS }
 | "/"    { DIV }
 | "*"    { TIMES }
 | "%"    { MOD }
 | "("    { LPAR }
 | ")"    { RPAR }
 (* identifiers *)
 | letter (letter | digit | '_')* as id { IDENT id }
 (* illegal characters *)
 | _ as c { failwith (Printf.sprintf "Illegal character '%c': " c) }
```

Lines 1 to 8 contain regular OCaml code that defines the token type and a function mk_int that builds an integer from a string. The program fails if the string is not a correct integer. Can it happen? Lines 9 to 12 define regular expressions to be reused in the rules. The remaining lines define the only rule of the file, named token, that recognizes an EXPR token. As long as there is newlines characters or blank characters, the lexer continues its reading of the input buffer. If we reach the end of the buffer (special character eof), we return the token representing it, EOF. When we recognize a token, we return it and if an unknown character (one for which no rule matches) is encountered, we fail with an error.

This file[3] can be compiled by `ocamllex`. A file `exprLexer_standalone.ml` is then generated. In this file the type token is defined, so do the functions mk_int of type `string -> token` and token of type `Lexing.lexbuf -> token` implementing the defined automaton.

You can test it using `utop`:

```
# let add i = i + 10 ;;
val add : int -> int = <fun>
```

```
utop # #use "exprLexer_standalone.ml" ;;
type token =
    EOF
  | PLUS
  | MINUS
  | TIMES
  | DIV
  | MOD
  | LPAR
  | RPAR
  | INT of int
  | IDENT of string
```

```
val mk_int : string -> token = <fun>
```

```
val token : Lexing.lexbuf -> token = <fun>
```

```
utop # let buffer = Lexing.from_string "13 + x1" ;;
val buffer : Lexing.lexbuf =
  {Lexing.refill_buff = <fun>; lex_buffer = Bytes.of_string "13 + x1";
  lex_buffer_len = 7; lex_abs_pos = 0; lex_start_pos = 0; lex_curr_pos = 0;
  lex_last_pos = 0; lex_last_action = 0; lex_eof_reached = true;
  lex_mem = [||];
  lex_start_p =
  {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0};
  lex_curr_p =
  {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0}}
```

```
utop # token buffer ;;
- : token = INT 13
```

```
utop # token buffer ;;
- : token = PLUS
```

```
utop # token buffer ;;
- : token = IDENT "x1"
```

```
utop # token buffer ;;
- : token = EOF
```

[3]It can be downloaded from moodle.

2.3 Precisions

An Ocamllex rule can be considered as a function because `ocamllex` generates a function with the same name. This function is recursive as we have seen in the previous example (token calls token when eliminating spaces and newlines).

You can add parameters to a rule. We will illustrate it with a small example that probably would not require the use of Ocamllex. The objective is to write an automaton to count the occurrences of the character 'a' in a string. For this, we define a rule which has as parameter (value) containing the number of 'a' already met. When the automaton encounters a 'a', it calls itself recursively with a value incremented by 1 (line 3). When the string ends, the automaton returns the number of 'a' (line 4). The desired function is then defined in the postlude reusing the rule (line 6).

```
1 rule count value = parse
2  | [^'a']* { count value lexbuf }
3  | 'a'   { count (value + 1) lexbuf }
4  | eof   { value }
5 {
6  let count_a s = let buffer = Lexing.from_string s in count 0 buffer
7 }
```

Once compiled by `ocamllex` and loaded in the interpreter, the function can be tested.

```
utop # #use "count_a.ml" ;;
```

```
val count : int -> Lexing.lexbuf -> int = <fun>
```

```
val count_a : string -> int = <fun>
```

```
utop # count_a "eratatata" ;;
- : int = 4
```

⚠ ocamlbuild

Now and for the rest of the UV, you should use `ocamlbuild` to compile and directly produce an executable file. You need to add the following line to bind the definition of the main function (here compile)[4].

```
let _ = Arg.parse [] compile ""
```

3 Parsing with Menhir

The Menhir files use the extension `.mly` and have the following form:

```
%{
 (* OCaml code *)
%}
 (* Declarations of symbols *)
%%
 (* Rules *)
%%
 (* OCaml code *)
```

The main structure is similar to Ocamllex with a prelude and a postlude, some declarations and a set of rules.

[4]For more details, do not hesitate to consult the manual section on the module Arg.

3.1 The declarations

The following declarations are possible:

- `%token` (< *type* >)? $symbol_1$ ... $symbol_n$: defines the n symbols as lexical tokens. They are added as constructors to the type `token`. When a type is given the n constructors take it as argument. By consequences the lexer does not need to define the token type anymore, instead it uses the parser's one.
- `%start` (< *type* >)? $symbol_1$ ... $symbol_n$: defines the n symbols as entry points. A parsing function of the same name is defined for each symbol. The symbol must be a non terminal left part of a rule. The return type of this function can be given simultaneously or using the following declaration.
- `%type` (< *type* >)? $symbol_1$ ... $symbol_n$: defines the return types of the actions corresponding to the n symbols. Each symbol must be a non terminal left part of a rule. These type declarations are only mandatories for entry points.
- the priority and associativity of symbols:
  - `%left` $symbol_1$ ... $symbol_n$
  - `%right` $symbol_1$ ... $symbol_n$
  - `%nonassoc` $symbol_1$ ... $symbol_n$

  The name specifies the associativity and the order of appearance in the file specifies the priority. The first defined has the weakest priority. When on the same line they share associativity and priority. Associativity and priority are used when there is a conflict. For example, the expressions $1+2*3$ can be recognized (depending on the defined rules) as $(1+2)*3$ or $1+(2*3)$. First, the parser use priority. Here for example, if $*$ has a higher priority than $+$, the second form is adopted. In case of similar priority, it uses associativity. The first term corresponds to left and the second to right. If the symbol is declared non associative, a parsing error is raised. Often binary operators are left associative and unary ones are right associative.

3.2 The rules

The rules follow the syntax:

```
nonterminal:
 | symbol ... symbol { (* semantic action *) }
 | ...
 | symbol ... symbol { (* semantic action *) }
```

A semantic action contains OCaml code that builds and returns the semantic value of the non-terminal in the corresponding case. This semantic action can use the semantic value of all terminals and non-terminals that appear in the corresponding production. Two ways to access their value are available: (1) by naming these symbols in production or (2) by position `$1` for the first symbol and up to `$9`. The second possibility is deprecated because it creates a strong coupling between action and production (if a symbol is moved, the action code must be changed). In general, the semantic action consists in constructing the node of the Abstract Syntax Tree associated with the recognized sentence.

To demonstrate how Menhir works, we're going to examine an example in details. Let's look at a parser for Expr.

```
1  %{
2   open ExprAst
3   open BinOp
4  %}
5  %token EOF PLUS MINUS TIMES DIV MOD LPAR RPAR
6  %token <int> INT
7  %token <string> IDENT
8  %start < ExprAst.expression > expression
9  %left PLUS MINUS
10 %left TIMES DIV MOD
11 %right UMINUS
12 %%
13 expression:
14  | e=expr EOF        { e }
15 expr:
16   | MINUS e=expr %prec UMINUS { Uminus e }
17   | e1=expr o=bop e2=expr    { Binop(o,e1,e2) }
18   | e=simple_expr        { e }
19 simple_expr:
20   | LPAR e=expr RPAR       { e }
21   | id=IDENT             { Var id }
22   | i=INT              { Const i }
23 %inline bop:
24   | MINUS    { Bsub }
25   | PLUS    { Badd }
26   | TIMES    { Bmul }
27   | DIV     { Bdiv }
28   | MOD     { Bmod }
29 %%
```

Lines 2 and 3 open modules to make them available inside actions. Lines 5 to 7 define the various tokens. The two last tokens `INT` and `IDENT` can carry information. More precisely, a token of type `INT` will contain the corresponding integer and the token `IDENT` contains the string of the identifier. Line 8 defines the only entry point: the non terminal `expression`, its type corresponds to the AST defined for expressions. Lines 9 to 11 define priorities and associativities. `UMINUS` has a higher priority than `TIMES`, `DIV` and `MOD` that have a higher priority than `PLUS` and `MINUS`. Notice here that the symbol `UMINUS` is not defined using `%token`. Such a symbol will have to be used by a `%prec` in a rule (here on line 16). Lines 13, 15 and 21 define the rules of derivation for three non terminals. If we look in more details line 17, an `expr` can be derived as an `expr` followed by a `bop` and followed by an `expr`. The result of the action is a term `Binop` collecting the content of these three elements using variables.

When compiling this file, Menhir will produce an OCaml module of signature `exprParser.mli` and of implementation `exprParser.ml`. This module defines:

- the type `token`:

  ```
  type token =
    | TIMES
    | RPAR
    | PLUS
    | MOD
    | MINUS
    | LPAR
  ```

```
| INT of (int)
| IDENT of (string)
| EOF
| DIV
```

- an exception for parsing errors `Error`
- a parsing function named `expression` of type `(Lexing.lexbuf -> token) -> Lexing.lexbuf -> exprAst.expression` that takes as argument, the lexer, the input buffer and returns an AST expression.

The lexer corresponds to the code already presented in section 2.2 without the definition of the type `token` which is now defined by the parser.

Remark:

The signature does not include the prelude of the Menhir specification file. So any types that the signature will contain must be fully qualified. (using the dot notation and the name of their module). By consequence, all the types of entry points and the types parameterizing a token must be fully qualified (as in line 8 above).

PC3, TP7-8 – Let's practice compilation

Objectives

At the end of the activity, you should be capable of:

- to define the syntax and implement the corresponding lexer and parser for a simple language,
- give the formal semantics of a simple language,
- implements a simple interpreter of it,
- implements a simple translation from one language to another one.

Part I

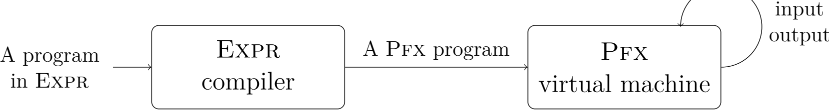
Context

1 The big picture

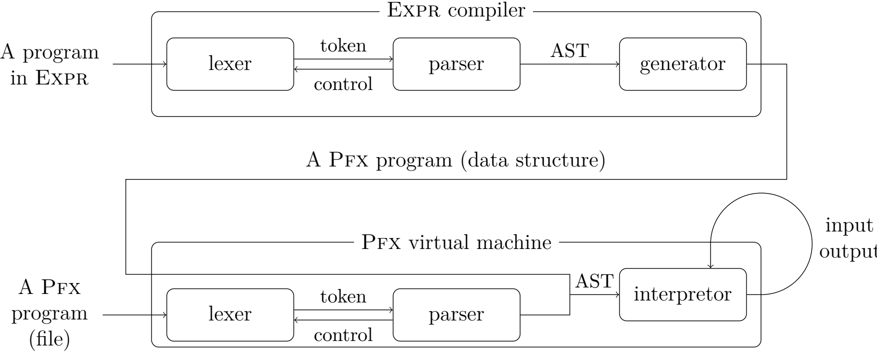
During our practice of compilation, we are going to write a compiler for a simple arithmetic expression language. This simple language is named Expr and follow the exercise 6 directly from the *Discovering OCaml* document.

To make it simpler for you, instead of generating X86 code to execute it on a standard machine, we are going to use a *virtual machine*. Such a virtual machine is a program that takes a low-level program and executes it. We talk about interpretation. For pedagogical reason, we stick to a simple and minimal low-level language. This low-level language is called Pfx and is a so-called stack language (this will become clear when we will define it).

To summarise, we are going to work on two components:



These two components, the Expr compiler and the Pfx virtual machine follow a similar architecture: first they parse their input language to produce an internal data structure (the AST), and then treat this AST. However, they differ in their treatment. The Expr compiler produces a corresponding Pfx program while the Pfx virtual machine *executes* its input.



For pedagogical reason and testing purpose, the Pfx virtual machine is able to receive a (Pfx) AST directly or to parse a file containing a Pfx program.

2 Work to be done

The overall objective is to produce a working Expr compiler and a working Pfx virtual machine.

The work will happen both at a mathematical (conceptual) level and at a programming level, the first being the specification of the second. You will be required to answer three kinds of questions:

- expl: you are supposed to produce a textual explanation
- math: you are supposed to give a set of mathematical definition, often in the form of a set of inference rules
- code: you are supposed to produce code; as usual, this code should be of quality (well indented, commented, with a good choice of identifiers, tested, . . .)

The work is organised in a set of questions you are supposed to work on linearly. Moreover, at every step, you will have both a working Expr compiler and a working Pfx virtual machine. All along the subject, we enrich both the user language Expr and the low-level language Pfx.

During your work, you will need:

- the document containing lecture notes about *Compilation with OCaml* (`notes.pdf`). To succeed in writing your compiler, **the reading of this document is necessary**: it gives hints related to the compilation process, to the tools you will use (`ocamllex` and `menhir`). It also gives you the syntax and links to the official documentation.
- this document presenting the questions and the work to be done.

We also provide you an additional document to help you to remember the tools and commands: `cheatsheet.pdf`.

If you are proficient in OCaml and in the compilation field, just read the current document (please note that we also extend the language in order to handle simple functions and closures), then shut up and hack!

Up to the exercise you reach, all questions have to be answered. Code-related ones are answered by writing OCaml code and a quick note somewhere (comments within the code, quick summary in another file such as the README, etc.). Other questions have to be answered in a separated file, using an interoperable format (plain text or PDF for instance). It is obviously possible to answer mathematical questions by learning and using LaTeX, however this is not the point of this course (and could be time-consuming...). An efficient way to answer such questions is to scan a *readable* handwritten answer.

3 How and what to deliver?

To help you with your deliverable, we provide a project skeleton (`project_skeleton.tar.gz`) which should be renamed, adapted and extended following your needs. Please also read the README example.

We expect you to deliver a compressed archive (`.tar.gz` file[1]) containing:

- a PFX virtual machine written in OCaml, using `ocamllex` and `menhir`, wrt the specifications that are expressed throughout the questions,
- an EXPR compiler written in OCaml, using `ocamllex` and `menhir`, wrt the specifications that are expressed throughout the questions,
- a README (plain text file) explaining your work.

To be sure to be evaluated and to hope to obtain a strictly positive mark, you should also pay close attention to the form of your deliverable:

- deliver only one compressed archive (`tar.gz format`) of your top level directory (meaning that decompressing it should result in a single directory),
- please respect the following naming convention: `NAME1-NAME2.tar.gz`,
- file encoding: UTF-8 (especially if you write non-ASCII characters...),
- clean your directory: remove all useless files and files that can be generated (for example the `_build` directory),
- choose meaningful names (file names and contents) understandable by other people,
- comment your code,
- indent your code consistently,
- organise your files (use subdirectories if needed),
- join a README file and answers to the non-code-related questions.

[1] `tar.gz` is not `zip`, `rar`, `7z`, `tar.xz`, `tar.bz2`, etc. and should be built using the `tar` tool.

Part II
Questions

4 A simple stack language

During our practice of compilation, we are going to use a stack language: PFX. As expressed by its name, it is a language in the tradition of Postscript[2] and Forth[3] relying on a stack to store value instead of variables. In such a language, all operations act on the elements of the stack.

Exercise 1 (*expl*)

▷ **What is a stack? What are the operations that you usually execute on a stack?**

PFX is inspired by the Postfix machine of the book "Design Concepts in Programming Languages"[4]. It is intentionally kept simple. Whenever, you are not satisfied with it, feel free to extend and modify it... But not during the lab sessions.

4.1 Informal description

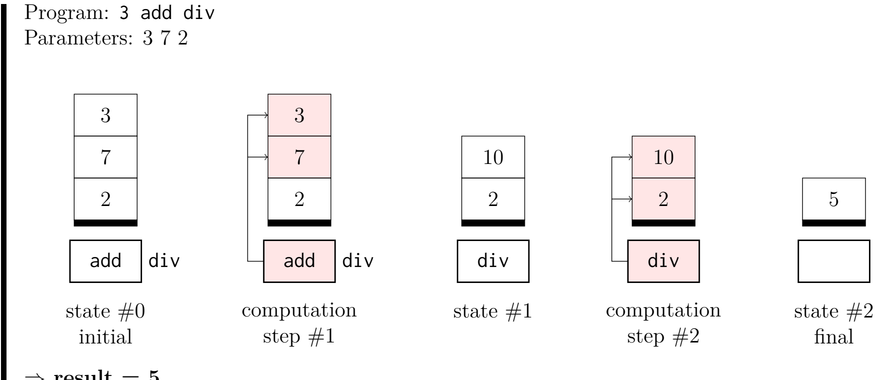
A PFX program begins with an integer specifying the number of arguments it will need to be run, then it is composed of a sequence of basic instructions. Basic instructions are `push`, `pop`, `swap` (it exchanges the first two elements of the stack) and the five arithmetic operations `add`, `sub`, `mul`, `div` and `rem`. Only `push` takes an argument which is an integer. All the arithmetic operations behave similarly: they use the first two elements of the stack as arguments, remove them and push back the result. This first version only manipulates integers so all values are integers.

For example, the program `0 push 12 push 7 sub` returns -5 while `0 push 12 push 7 swap sub` returns 5.

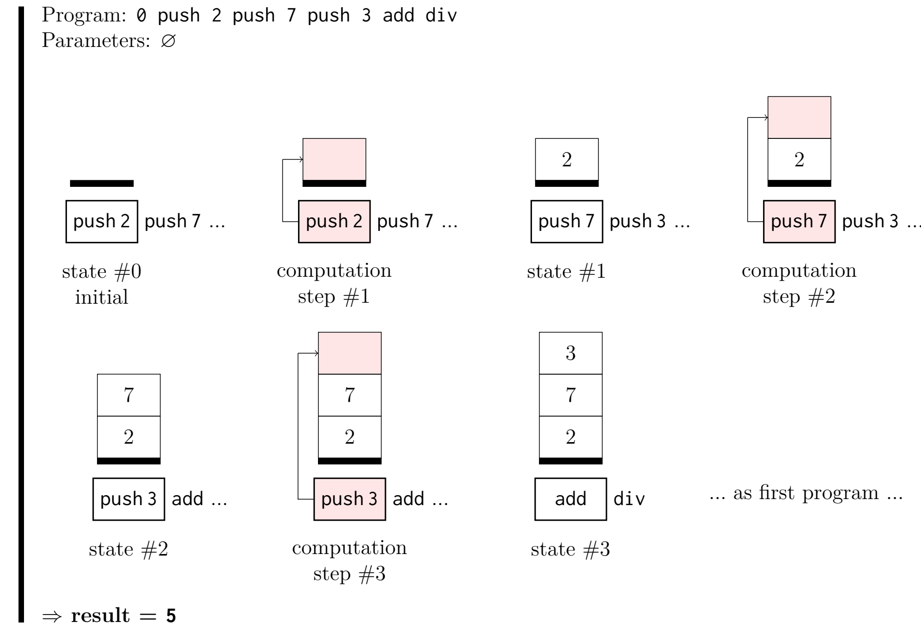
Program arguments are pushed onto the stack from last to first, before the execution of the program commands.

Here follow two examples of a program execution in detail.

[2] `https://en.wikipedia.org/wiki/PostScript`
[3] `https://en.wikipedia.org/wiki/Forth_(programming_language)`
[4] `https://mitpress.mit.edu/books/design-concepts-programming-languages`



The first example is equivalent to the following one:



Exercise 2 (*expl*)

▷ **Detail in the same way the execution of `0 push 12 push 7 swap sub`.**

4.2 Formal semantics

To formalise the semantics of Pfx, we will use the following notations:

- a program is a pair i, Q where i is the number of awaited arguments and Q the sequence of instructions composing the program, the empty instruction sequence is written $\varnothing$,
- an instruction sequence is built by the constructor '.', the sequence composed of I then Q is $I.Q$,
- a stack is built by the constructor '::', adding an element n to the stack S is $n :: S$, the empty stack is also written $\varnothing$, the length of the stack S can be obtained by $\#S$,

The semantics of Pfx uses two sets of rules, one for programs that specify the execution of the complete program and one to describe each possible computational step on a current instruction sequence and the current stack. The rules for programs use the rule for computational steps.

Exercise 3

For programs, we have the following semantics:

$$(1)\ \frac{i \neq n}{v_1, \ldots, v_n \vdash i, Q \Rightarrow \textsc{Err}} \qquad (2)\ \frac{Q, v_1 :: \ldots :: v_n :: \varnothing \rightarrow^* \textsc{Err}}{v_1, \ldots, v_n \vdash n, Q \Rightarrow \textsc{Err}}$$

$$(3)\ \frac{Q, v_1 :: \ldots :: v_n :: \varnothing \rightarrow^* \varnothing, v :: S}{v_1, \ldots, v_n \vdash n, Q \Rightarrow v}$$

The reduction rule $\rightarrow$ specifies the small step semantics of instructions and $\rightarrow^*$ its transitive closure[5]. $Q, S \rightarrow Q', S'$ means that in one step the execution of instruction sequence Q with stack S leads to instruction sequence Q' and stack S'.

▷ **Question 3.1 (*expl*):**
Explain using plain words the semantics of programs.

▷ **Question 3.2 (*math*):**
A case is still missing, spot it out and give the corresponding rule.

▷ **Question 3.3 (*math*):**
Give the rules describing the small step semantics for instruction sequences. Beware to cover all cases of runtime errors.

4.3 Implementation

Exercise 4

▷ **Question 4.1 (*code*):**
Propose the OCaml code for a type `command` describing the Pfx instructions. It should be in the file `pfxAst.ml`.

[5]The rule is potentially applied several times in sequence. Mathematically, $A \rightarrow^* B$ if and only if $A = B$ or there exists C such that $A \rightarrow C$ and $C \rightarrow^* B$.

▷ **Question 4.2 (*code*):**
Write an OCaml function `step` that implements the small step reduction you defined above the Pfx instructions. It should be in the file `pfxEval.ml`.

⚠ You should test your code by running some PFX programs defined as OCaml values (not yet parsed from files, this will be done later).

5 A simple arithmetic expression language

Following on the exercise 6 from the *Discovering OCaml* document, we will define a simple arithmetic expression language, named EXPR in order to compile it to PFX.

This first version of EXPR AST is implemented in file `exprAst.ml` as follows:

```
type expression =
  | Const of int
  | Var of string
  | Binop of BinOp.t * expression * expression
  | Uminus of expression
```

where the type `BinOp.t` is implemented in file `binOp.ml` as follows:

```
type t =
  | Badd | Bsub | Bmul | Bdiv | Bmod
```

You can explore the code provided within the project skeleton.

Exercise 5

▷ **Question 5.1 (*math*):**
Propose a compilation schema of Expr in Pfx. Give its formal description. Notice that with the current definition of Pfx, we cannot implement variables. We defer their implementation to a later exercise.

▷ **Question 5.2 (*code*):**
Define a function `generate` implementing the semantics you defined in previous question. It should be in the file `exprToPfx.ml`.

⚠ At this point, you have a first working compiler of EXPR and you should be able to execute simple programs.

6 Parsing

Read the document containing the lecture notes entitled *Compilation with OCaml* (`notes.pdf`) in order to understand `ocamllex` and its syntax. You can explore the code for the lexer and the parser provided within the project skeleton.

Exercise 6 (*A first* Pfx *lexer*)

▷ **Question 6.1 (*code*):**
Write a lexer for the Pfx stack machine language. Complete the provided `pfxLexer.mll`. To test it without the parser, have a look at the file `exprLexer_standalone.mll` on Moodle.

The following OCaml code[6] provides a way to read the string to parse from a file. The name of the file is given as an argument on the command line and is automatically passed to the function `compile`:

```
(* Entry point of the program, should contain your main function: here it is
 named parse_eval, it is the function provided after question 6.1 *)
(* The arguments, initially empty *)
let args = ref []
(* The main function *)
let parse_eval file =
 print_string ("File "^file^" is being treated!\n");
```

▷ **Question 6.2 (*code*):**
Reuse this code to be able to parse a file containing a Pfx program and prints all the tokens encountered in the process.

⚠ You should test your lexer and use test files.

Exercise 7 (*Locating errors, code*)

Generally, a compiler should be able to return an error message containing the location of the error to its user. OCaml module `Lexing` defines a type `position` for this purpose.

```
type position = {
 pos_fname : string; (* name of the file *)
 pos_lnum : int;     (* number of the line *)
 pos_bol : int;      (* nb of chars between the beginning of the file and the one of current line *)
 pos_cnum : int;     (* nb of characters since the beginning of the file *)
}
```

By default, the generated lexer only updates the last element (`pos_cnum`). The actions must take care of the others. The functions `lexeme_start_p` and `lexeme_end_p` of the module `Lexing` allows one to get respectively the location of the beginning and the end of the current token. To help you, we provide the module `Location`[7] which defines helpful elements. For the moment, you should only use:

- the exception `Location.Error` carrying both a message and the location of the error;
- the type `Location.t` of a location composed of a starting position and an ending position;
- the function `Location.init` setting the file name of the given buffer;
- the function `Location.incr_line` increasing a line in the given buffer;
- the function `Location.curr` return the current position of the given buffer;
- the function `Location.print` printing the given location.

▷ **Modify your code from the previous exercise to be able to return the location of errors.**

⚠ Now and for the rest of the UV, you should use `ocamlbuild` to compile and directly produce an executable file. You need to add the following line to bind the defined main function (here `compile`)[8].

[6] Provided in the project skeleton, on Moodle.
[7] As always, available on Moodle.
[8] For more details, do not hesitate to consult the manual section on the module `Arg`.

```
let _ = Arg.parse [] compile ""
```

Exercise 8 (*A first* Pfx *parser*)

▷ **Question 8.1 (*code*):**
Write a parser for the Pfx stack machine language.

▷ **Question 8.2 (*code*):**
Test it in combination with your Lexer. To do it, you will have to write a function that prints the AST of Pfx. You should now use the provided file `pfxVM.ml` as the main file for the Pfx virtual machine. It is the file that should be given to `ocamlbuild` as a target.

Notice that it requires that you modify slightly your lexer to remove the main functions and replace the token type definition by an open of the parser module.

⚠ You should test with more than one test your Pfx parser!

7 Simple functions

Let's suppose we would like to add notions of function and application to Expr. One way of doing it is to add a definition of function expression and an application expression. As in the λ-calculus, we limit ourselves to function with a unique argument. This leads to the following AST.

```
type expression =
  | Const of int
  | Var of string
  | Binop of BinOp.t * expression * expression
  | Uminus of expression
  (* For function support *)
  | App of expression * expression
  | Fun of string * expression
```

⚠ You can find the new AST and the modified lexer and parser in the directory `expr_fun` of the skeleton. You can replace the corresponding files in `expr` by their new version from `expr_fun`.

Currently, Pfx cannot be used to generate code including functions. We first have to modify the language Pfx. Three new instructions must be added:

- executable sequence `(` Q `)`, where Q is an usual instruction sequence, when it is encountered the executable sequence is pushed on the top of the stack;
- `exec` which is an instruction that pops the top of the stack and executes it by appending it in front of the executing sequence, notice that the top of the stack must be an executable sequence;
- `get` pops the integer i on top of the stack, and copies on top of the stack the i-th element of the stack, it raises an error if there is not enough element on the stack.

Notice that these new constructions impose that the stack now contains two kinds of objects: integer or executable sequence.

Be sure to understand that Pfx does not have functions, it only has executable sequence.

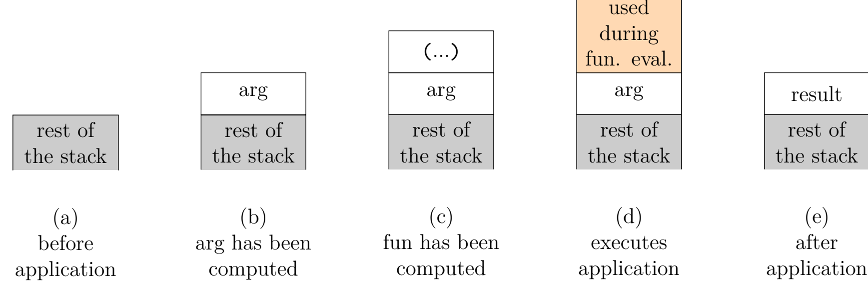


Figure 1: Usage of the stack during application

Exercise 9

▷ **Question 9.1 (*expl*):**
Do we need to change the rules for the already defined constructs?

▷ **Question 9.2 (*math*):**
Give the formal semantics of these new constructions.

▷ **Question 9.3 (*code*):**
If needed, extend the lexer and parser of Pfx to include these changes.

The translation of Expr to Pfx must be revised. The idea is that a λ-abstraction is translated to an executable sequence and an application is translated to an `exec` plus some code to *clean up* the stack. This executable sequence supposes that its parameter is on the top of the stack when it starts to execute. In its body, whenever it wants to use its parameter it `get`s it from where it is using an environment $\mathcal{P}$ associating a variable to its position in the stack (its depth). Consequently, during the translation, we have to keep track of the depth at which each parameter is. When application terminates, it pops out the parameter from the stack. This behaviour is described in the figure 1:

(a) the stack is in some state

(b) a computation pushes the argument of the future call

(c) another computation pushes the function to call, it must be an executable sequence

(d) the call is made, probably using the stack and its argument

(e) the call ends and pushes its results on the stack

Beware that during step (d), the stack may grow and therefore each time something is pushed onto the stack we need to update the position of the various reachable arguments.

Exercise 10

▷ **Question 10.1 (*expl*):**
Give the compiled version of the expression $(\lambda x.x + 1)\ 2$**. Then describe step by step the evaluation of its Pfx translation.**

▷ **Question 10.2 (*math*):**

Give the formal rule for transformation.

▷ **Question 10.3 (*code*):**
Provide a new version of `generate`.

▷ **Question 10.4 (*expl*):**
Give the compiled version of the expression $((\lambda x.\lambda y.(x-y))\ 12)\ 8$. Then describe step by step the evaluation of its Pfx translation. What do you think of the result? What is happening?

A straight forward extension of the syntax consists in adding a so-called *syntactic sugar* to support `let` definition (as of OCaml).

Exercise 11 (*Syntactic sugar*)

▷ **Question 11.1 (*expl*):**
What is the translation in the syntax of Expr already defined of a new `let` $x = e_1$ `in` e_2?

▷ **Question 11.2 (*code*):**
What part of the code must be modified to get support for `let`?

8 Closure

To solve the problem illustrated in question 10.4, we need to change the function value. When defining a function value, we need to capture and store all its free variables (the variables defined by its context). Indeed, the function may be executed in a different scope that the one where it was defined, so it must be able to retrieve the values of variables of its definition location when executed. This new kind of value is called a *closure* and it is a pair composed of an environment (of the free variables of the function) and the function. So the closure of a function $\lambda x.e$ in the context of an environment $\mathcal{E}$ is denoted $\{\mathcal{E}, x, e\}$. Notice that the domain of $\mathcal{E}$ is limited to set of free variables of $\lambda x.e$. When this domain is empty, we fall down on the function value of the previous section. This explains why and when it was working!

The formal semantics of EXPR is presented in figure 2. The values may be integer from set $\mathbb{I}$, closures (set $\mathbb{C} = X \rightarrow (\mathbb{I} \cup \mathbb{C}) \times X \times Expr$[9]) or error (value ERR). All operations are extended to produce an error whenever they are applied to an error (*e.g.* $-\text{ERR} = \text{ERR}$ and $\text{ERR} + v = \text{ERR}$).

Exercise 12 (*math*)

▷ **Give the proof derivation computing the value of the term of question 10.4 ($((\lambda x.\lambda y.(x-y))\ 12)\ 8$).**

[9] X is the set of variables and $Expr$ the set of terms of EXPR.

$$(\textsc{Const})\ \mathcal{E} \vdash v \Rightarrow v \qquad (\textsc{Var})\ \frac{x \in dom(\mathcal{E})}{\mathcal{E} \vdash x \Rightarrow \mathcal{E}(x)} \qquad (\textsc{VarErr})\ \frac{x \notin dom(\mathcal{E})}{\mathcal{E} \vdash x \Rightarrow \textsc{Err}}$$

$$(\textsc{Uminus})\ \frac{\mathcal{E} \vdash e \Rightarrow v}{\mathcal{E} \vdash -e \Rightarrow -v} \qquad (\textsc{Binop})\ \frac{op \in \{+,-,*\} \quad \mathcal{E} \vdash e_1 \Rightarrow v_1 \quad \mathcal{E} \vdash e_2 \Rightarrow v_2}{\mathcal{E} \vdash e_1\ op\ e_2 \Rightarrow v_1\ op\ v_2}$$

$$(\textsc{Div})\ \frac{op \in \{/,\%\} \quad \mathcal{E} \vdash e_1 \Rightarrow v_1 \quad \mathcal{E} \vdash e_2 \Rightarrow v_2 \quad v_2 \neq 0}{\mathcal{E} \vdash e_1\ op\ e_2 \Rightarrow v_1\ op\ v_2}$$

$$(\textsc{DivErr})\ \frac{op \in \{/,\%\} \quad \mathcal{E} \vdash e_2 \Rightarrow 0}{\mathcal{E} \vdash e_1\ op\ e_2 \Rightarrow \textsc{Err}}$$

$$(\textsc{App})\ \frac{\mathcal{E} \vdash e_1 \Rightarrow \{\mathcal{E}_c, x, e\} \quad \mathcal{E} \vdash e_2 \Rightarrow v_2 \quad \mathcal{E}_c, x \mapsto v_2 \vdash e \Rightarrow v}{\mathcal{E} \vdash e_1\ e_2 \Rightarrow v}$$

$$(\textsc{AppErr1})\ \frac{\mathcal{E} \vdash e_1 \Rightarrow v_1 \quad v_1 \notin \mathbb{C}}{\mathcal{E} \vdash e_1\ e_2 \Rightarrow v_1\ op\ v_2} \qquad (\textsc{AppErr2})\ \frac{\mathcal{E} \vdash e_1 \Rightarrow \{\mathcal{E}_c, x, e\} \quad \mathcal{E} \vdash e_2 \Rightarrow \textsc{Err}}{\mathcal{E} \vdash e_1\ e_2 \Rightarrow \textsc{Err}}$$

$$(\textsc{Fun})\ \frac{dom(\mathcal{E}') = dom(\mathcal{E}) \quad \forall x \in dom(\mathcal{E}), \mathcal{E}'(x) = \mathcal{E}(x)}{\mathcal{E} \vdash \lambda x.e \Rightarrow \{\mathcal{E}', x, e\}}$$

Figure 2: Big step operational semantics of EXPR

Exercise 13

▷ **Question 13.1 (*expl*):**
Is it possible to translate Expr to Pfx? If yes, can you give the idea of the translation. If no, what would be necessary to add to Pfx ?

The chosen implementation of closure we are going to explore is the following.

A closure is an executable sequence that begins with instructions pushing onto the stack the values of the free variables. Notice that this part of the executable sequence must be added at runtime (the only moment when the values are known).

To enable the modification of an executable sequence at runtime we add a new construct to PFX: `append`. This construct expects the stack to contain a value on the top and an executable sequence below. It appends the command storing the "value" in the stack at the beginning of the executable sequence. When the value is an integer, this command is a push of it. The command is the value when it is an executable sequence. It also adds the operation to remove this value from the stack when the function ends.

▷ **Question 13.2 (*math*):**
Give the formal semantics of `append`.

▷ **Question 13.3 (*code*):**
If needed, extends the lexer and parser of Pfx to include these changes.

▷ **Question 13.4 (*math*):**
Give the formal rules of translation from Expr to Pfx to support closure.

▷ **Question 13.5 (*code*):**
Provide a new version of `generate`.

▷ **Question 13.6 (*expl*):**
Give the compiled version of the expression $((\lambda x.\lambda y.(x - y))\ 12)\ 8$. Then describe step by step the evaluation of its Pfx translation. Is it better?

9 Extensions

To explore compilation in more depth, complete the following extensions by giving the formal version and then by implementing them. For each extension, a piece of information on the difficulty is provided.

1. (*regular*) Extend the stack machine with a `sel` instruction that requires a stack of at least three elements. Its result is the third element of the stack if the first is 0 and the second in the other case. Use this extension to add booleans, boolean operators, comparison operators and a condition operator $e_1 ? e_2 : e_3$ similar to the one of Java. Notice that the game is to try to add as little new instructions to Pfx as possible[10].

2. (*medium*) Extend EXPR with a `let rec` construct enabling the definition of recursive functions. *Hint*[11]. Propose a mechanism to compile it to PFX. *Hint*[12].

3. (*challenging*) Extend PFX with mechanisms to manipulate the stack. In this extension, a new value can be on the stack, a stack. Add the following constructs:

   - `pack` pushes a stack value containing current stack content after clearing it;
   - `unpack` pops the top of the stack which must be a stack value and replaces current stack by this value;
   - `switch` pops the top of the stack which must be a stack value, packs the rest of the current stack, replaces current stack by the stack value and pushes rest stack value.

   These constructs provide so-called continuations. Use them to add to EXPR:

   (a) an exception mechanism,

   (b) basic threading

[10] My solution just adds one.

[11] **Do not return before trying!**

Define a recursive function f by $\mu f.\lambda v.e$ and a recursive closure $\{\mathcal{E}, f, x, e\}$. Evaluating such a recursive closure add it to the environment under the name f.

[12] **Do not return before trying!**

The easiest implementation is to define a notion of recursive instruction sequence that is kept on the stack when executed and only popped when terminating.

Quatrième partie

Logics

ELU 610
Langages et logique

Yannis Haralambous (IMT Atlantique)

29 novembre 2017

Première partie I

Introduction

Pour bien commencer

Tous les félins sont des mammifères.
Un chat est un félin.
Donc un chat est un mammifère.

Tous les félins sont des mammifères.
Un cabillaud n'est pas un mammifère.
Donc un cabillaud n'est pas un félin.

Modes de raisonnement

- La *déduction* : si A est vrai et si A vrai ⇒ B vrai, alors B est vrai.
- L'*abduction* : si B est vrai et si A vrai ⇒ B vrai, alors A est vrai. (Exemple : les ânes sont des êtres vivants, je suis un être vivant, donc je suis un âne.) Mais aussi : *elle consiste, lorsque l'on observe un fait dont on connaît une cause possible, à conclure à titre d'hypothèse que le fait est **probablement** dû à cette cause-ci* (Wikipédia).
- L'*induction* : si A et B sont vrais, alors A vrai ⇒ B vrai. (Exemple : la taille de pied des enfants et leur aptitude en orthographe.) Mais aussi : *genre de raisonnement qui se propose de chercher des lois générales à partir de l'observation de faits particuliers, **sur une base probabiliste*** (Wikipédia).

Pour bien commencer

ELU 610 *Langages et logique*

Yannis Haralambous (IMT Atlantique)

Introduction

Tous les félins sont des mammifères.
Un chat est un félin.
Donc un chat est un mammifère.

Tous les félins sont des mammifères.
Un cabillaud n'est pas un mammifère.
Donc un cabillaud n'est pas un félin.

Pour bien commencer

ELU 610 *Langages et logique*

Yannis Haralambous (IMT Atlantique)

Introduction

Toutes les blacies sont des chazomares.
Une élêthiote est une blacie.
Donc une élêthiote est une chazomare.

Toutes les blacies sont des chazomares.
Une exypnade n'est pas une chazomare.
Donc une exypnade n'est pas une blacie.

Pour bien commencer

ELU 610 *Langages et logique*

Yannis Haralambous (IMT Atlantique)

Introduction

Toutes les blacies sont des chazomares. $\forall X, P(X) \rightarrow Q(X)$
Une élêthiote est une blacie. $P(A)$
Donc une élêthiote est une chazomare. $\vdash Q(A)$

Toutes les blacies sont des chazomares. $\forall X, P(X) \rightarrow Q(X)$
Une exypnade n'est pas une chazomare. $\neg Q(B)$
Donc une exypnade n'est pas une blacie. $\vdash \neg P(B)$

Nos outils de travail

ELU 610 *Langages et logique*

Yannis Haralambous (IMT Atlantique)

Introduction

- L'intelligence humaine opère à travers des *processus de raisonnement* basés sur des *représentations (internes) de la connaissance*.
- En intelligence artificielle on crée des *bases de connaissances*. Celles-ci contiennent des *énoncés* exprimés dans des *langages de représentation des connaissances*.
- Il y a des énoncés de départ (appelés *axiomes*) et des processus de dérivation de nouveaux énoncés à partir des énoncés existants : l'*inférence*.
- On peut interroger une base de connaissances : pour obtenir la réponse à une requête, on se sert également du processus d'inférence.
- À la base de la représentation des connaissances, on trouve la logique.
- Il y a plusieurs types de logique : propositionnelle, du 1er ordre, du 2e ordre, modale, floue, temporelle, etc.

Notions communes à toutes les logiques

ELU 610 *Langages et logique*

Yannis Haralambous (IMT Atlantique)

Introduction

1. Une *syntaxe* du langage : ainsi, par exemple, $x + y = 4$ est un énoncé bien formé, $x4y+ =$ ne l'est pas.
2. Un *système déductif*, c'est-à-dire un ensemble d'énoncés qualifiés d'*axiomes* et une *méthode de preuve* qui permet d'en déduire d'autres énoncés (appelés *théorèmes*) à partir de ceux-ci, en utilisant des *règles d'inférence*.
3. Une *théorie des modèles*, c'est-à-dire une notion d'*interprétation* (ou de «monde possible ») des énoncés (intuitivement : une manière de leur donner du sens dans un domaine choisi) qui les dote d'une valeur de vérité. Une interprétation qui rend un énoncé vrai est un *modèle* de celui-ci (ainsi, $x + y = 4$ est vraie dans un monde où $x = y = 2$ et fausse dans un monde où $x = 1, y = -1$). On dit qu'un énoncé β est *conséquence* d'un énoncé α ($\alpha \models \beta$) quand tout modèle de α est aussi modèle de β. Exemple : $(x + y = 2) \models (x + y > 0)$.

Quel rapport entre preuve et conséquence ?

ELU 610 *Langages et logique*

Yannis Haralambous (IMT Atlantique)

Introduction

On montre, pour les logiques propositionnelle et du 1er ordre, que

- la preuve par déduction (dans le cadre d'un système déductif donné) et
- la conséquence (au sens de la théorie des modèles)

sont *équivalentes* : autrement dit, β est conséquence de α si et seulement si on peut déduire β à partir de α par un nombre fini d'inférences.

ELU 610 *Langages et logique*

Yannis Haralambous (IMT Atlantique)

Systèmes formels
La notion de système formel
Explications

Logique du 1er ordre

Deuxième partie II

Systèmes formels

Les systèmes formels

ELU 610 *Langages et logique*

Yannis Haralambous (IMT Atlantique)

Systèmes formels
La notion de système formel
Explications

Logique du 1er ordre

- Cette notion constitue le lien avec la première partie du module INF 424.
- Un *système formel* est la donnée :

1. d'un *alphabet* Σ fini ou dénombrable,
2. d'un sous-ensemble récursif F de l'ensemble Σ^* des suites finies d'éléments de Σ, on appelle F l'ensemble des *énoncés bien formés*,
3. d'un *système déductif*, c'est-à-dire
   1. d'un sous-ensemble récursif A de F, appelé ensemble des *axiomes*,
   2. d'un ensemble fini R de prédicats décidables définis sur F appelés *règles d'inférence*.

Explications

- Les *énoncés bien formés* F sont les combinaisons de symboles de Σ qui respectent une certaine syntaxe.
- À noter que l'ensemble des énoncés bien formés peut être défini par une grammaire formelle. ATTENTION ! ne pas confondre les règles de production de la grammaire formelle et les règles d'inférence du système formel !
- Les *règles d'inférence* permettent de déduire de nouveaux énoncés bien formés à partir d'énoncés existant.
- Les *axiomes* sont des énoncés que l'on a choisi de ne pas déduire d'autres énoncés, ils servent d'énoncés de départ à partir desquels on déduit tous les autres.

Explications

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Systèmes formels
La notion de système formel
Explications
Logique du 1er ordre

- Un élément r de R est une application de F^n (pour $n \geq 1$) dans l'ensemble {vrai, faux}. Au lieu d'écrire $r(f_1, f_2, \ldots, f_{n-1}, g)$ on écrira
$$\frac{f_1, f_2, \ldots, f_{n-1}}{g}\quad r$$
ou alors $f_1, f_2, \ldots, f_{n-1} \vdash_r g$, et on lira « g peut être déduit de $f_1, f_2, \ldots, f_{n-1}$ par la règle d'inférence r ».
- Une *preuve* dans un système déductif est une suite d'ensembles de formules telle que chaque élément est soit un axiome, soit peut être déduit par les éléments des ensembles précédents en appliquant les règles d'inférence. Si $\{A\}$ est le dernier ensemble de la suite, on dira que A est un *théorème*, et on écrira $\vdash A$.

Troisième partie III

Logique du 1er ordre

Rappel : signature

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Si Σ est un alphabet alors une *signature* est une application ar : $\sigma \to \mathbb{N}$ (appelée *arité*).
- On dit que $f \in \Sigma$ est *n-aire* si $\mathrm{ar}(f) = n$.
- Si $\mathrm{ar}(c) = 0$ on dira que c est une constante.
- L'*ensemble T_Σ des termes sur Σ* est la plus petite partie du monoïde libre Σ telle que
  1. si $c \in \Sigma$ alors $c \in T_\Sigma$,
  2. si $f \in \Sigma$ avec $\mathrm{ar}(f) = n$ et $M_1, \ldots, M_n \in T_\Sigma$, alors $f(M_1, \ldots, M_n) \in T_\Sigma$.
- Soit X un ensemble appelé *ensemble des variables*, alors L'*ensemble $T_\Sigma[X]$ des termes (avec variables) sur Σ* est la plus petite partie du monoïde libre Σ telle que
  1. si $c \in \Sigma$ alors $c \in T_\Sigma[X]$,
  2. si $f \in \Sigma$ avec $\mathrm{ar}(f) = n$ et $M_1, \ldots, M_n \in T_\Sigma$, alors $f(M_1, \ldots, M_n) \in T_\Sigma[X]$,
  3. si $x \in X$, alors $x \in T_\Sigma[X]$.

Système formel

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- La logique du 1er ordre est un *système formel* dont l'*alphabet* est constitué :
  - de *prédicats* n-aires P, Q, R, . . . ,
  - de *fonctions* n-aires $f, g, h, \ldots$,
  - de *variables* X, Y, Z, . . . ,
  - de *connecteurs* $\neg$ (négation) et $\to$ (implication),
  - du *quantificateur universel* $\forall$ («pour tout»),
  - du signe de l'*égalité* =,
  - et des *parenthèse*s ( ).

  Le cas particulier du prédicat 0-aire est appelé une *proposition*, et le cas de la fonction 0-aire est appelé une *constante*.

Système formel

- Parmi ces objets/symboles, certains ont une sémantique (= un sens) pré-établie, on les appelle des *symboles logiques* : $\neg$, $\to$, $\forall$, d'autres n'ont pas de sens en soi, et ne l'acquièrent que lors d'une *interprétation* (que l'on verra plus loin), on les appelle des *symboles non-logiques*.

Système formel

- La logique du 1er ordre a une signature fonctionnelle et une signature relationnelle. Les termes $T[X]$ sont définis par la signature fonctionnelle (constantes, fonctions, variables, qui produisent des termes fonctionnels).
- Les *formules* (on *énoncés*) sont basées sur les termes relationnels $R(t_1 \ldots t_n)$ où R appartient à la signature relationnelle et $t_i \in T[X]$. (cf. BNF du transparent suivant.)

Grammaire BNF de la logique du 1er ordre

Énoncé	$\to$	ÉnoncéAtomique | ÉnoncéComplexe
ÉnoncéAtomique	$\to$	Proposition | Prédicat(Terme+)
		| Terme = Terme
ÉnoncéComplexe	$\to$	(Énoncé)
		| ¬Énoncé
		| Énoncé → Énoncé
		| Quantificateur Variable, Énoncé
Terme	$\to$	Fonction(Terme+) | Constante | Variab
Quantificateur	$\to$	∀
Constante	$\to$	*A* | *B* | Michel. . .
Variable	$\to$	*X* | *Y* | *Z*. . .
Prédicat	$\to$	aime() | existe(). . .
Fonction	$\to$	père() | carré() | sinus(). . .

Précédence des opérateurs : $\forall$, $\neg$, =, $\to$.

Système formel

- Il faut bien comprendre que tant qu'on reste au niveau du système déductif (axiomes et règles d'inférence) et qu'on ne donne pas d'interprétation, on n'aura *que des symboles obéissant à certaines règles syntaxiques*.
- C'est la *théorie des modèles* qui nous donnera la possibilité d'*interpréter* ces symboles dans un domaine donné, aptes à être utilisés en mathématiques ou à modéliser la réalité.

Système déductif

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Rappel : un *système déductif* est un ensemble d'*axiomes* et un ensemble de *règles d'inférence*.

Axiomes

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Voici les axiomes de la logique du 1er ordre :
  1. $(P \to (Q \to P))$ (répétition),
  2. $(P \to (Q \to R)) \to ((P \to Q) \to (P \to R))$ (distribution conditionnelle),
  3. $(\neg P \to \neg Q) \to (Q \to P)$ (*contraposition*),
  4. $\forall x\, P(x) \to P(t)$, où t est un terme clos (élimination universelle),
  5. $\forall x\,(P \to Q) \to (\forall x\, P \to \forall x\, Q)$ (distribution universelle).

Règles d'inférence

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Rappelons la notation : si, à travers la règle d'inférence r, à partir de plusieurs énoncés $\alpha_1, \ldots, \alpha_n$ on peut inférer (= déduire, prouver) un énoncé β, alors on écrit

$$\frac{\alpha_1, \ldots, \alpha_n}{\beta}\, r.$$

- Deux règles d'inférence sont indispensables, la première est celle du *modus ponens* :

$$\frac{\alpha \to \beta \qquad \alpha}{\beta}\ \textit{modus ponens}$$

- et la deuxième est celle de la *généralisation* :

$$\frac{A}{\forall x\, A}\ \text{généralisation}.$$

Exemple

Montrons que $\forall x(P(x) \to P(x))$:

(1) $P(a) \to ((P(a) \to P(a)) \to P(a))$ — Ax. 1

(2) $P(a) \to ((P(a) \to P(a)) \to P(a)) \to ((P(a) \to (P(a) \to P(a)) \to (P(a) \to P(a))))$ — Ax. 2

(3) $(P(a) \to (P(a) \to P(a))) \to (P(a) \to P(a))$ — (1)+(2) *mod. pon.*

(4) $(P(a) \to (P(a) \to P(a)))$ — Ax. 1

(5) $P(a) \to P(a)$ — (3)+(4) *mod. pon.*

(6) $\forall x(P(x) \to P(x))$ — (5) général.

Ce n'est certes pas un résultat très spectaculaire, mais cela montre qu'il aurait été inutile de l'inclure parmi les axiomes.

Et maintenant, que fait-on ?

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Le *système déductif* nous fournit un formalisme. Nous «jouons» avec les symboles : nous formons des énoncés bien formés, nous déclarons que certains parmi eux sont des axiomes et nous obtenons d'autres énoncés par inférences successives. Mais à quoi cela sert-il ?
- Pour faire le lien avec les mathématiques ou le monde réel, il faut donner un sens aux symboles, et en déduire des valeurs de vérité pour les énoncés dans des domaines d'application donnés. Cela se fait à travers la *théorie des modèles*.
- Mais avant d'attaquer cette dernière, définissons quelques connecteurs supplémentaires dans le but de rapprocher le formalisme des énoncés de la langue naturelle.

Symboles supplémentaires

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Pour faciliter la lecture des énoncés, ainsi que le passage des situations du monde réel aux énoncés logiques, on introduit trois connecteurs supplémentaires : le $\wedge$ («et», *conjonction*), le $\vee$ («ou», *disjonction*) et le $\leftrightarrow$ («si et seulement si», *double implication*), ainsi que le quantificateur existentiel $\exists$ («il existe»).
- Ils sont définis à partir des symboles existants de la manière suivante :

- $A \vee B = \neg B \to A$,
- $A \wedge B = \neg(\neg A \vee \neg B) = \neg(B \to \neg A)$,
- $A \leftrightarrow B = (A \to B) \wedge (B \to A) = \neg((B \to A) \to \neg(A \to B))$,
- $\exists x\, P = \neg \forall x \neg P$.

Grammaire BNF enrichie

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

Énoncé	→	ÉnoncéAtomique | ÉnoncéComplexe
ÉnoncéAtomique	→	Proposition | Prédicat(Terme+)
		| Terme = Terme
ÉnoncéComplexe	→	(Énoncé)
		| ¬Énoncé
		| Énoncé ∧ Énoncé | Énoncé ∨ Énoncé
		| Énoncé → Énoncé
		| Énoncé ↔ Énoncé
		| Quantificateur Variable, Énoncé
Terme	→	Fonction(Terme+) | Constante | Variab
Quantificateur	→	∀ | ∃
Constante	→	*A* | *X* | Michel...
Variable	→	*a* | *x* | *s*...
Prédicat	→	aime | existe...
Fonction	→	Père | Carré | Sinus...

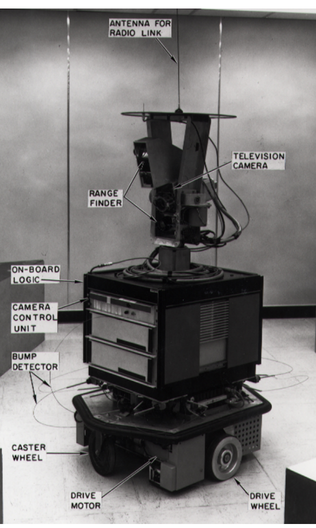
Précédence des connecteurs : $\forall/\exists, \neg, =, \wedge, \vee, \to, \leftrightarrow$.

Variables libres et liées

- Une variable est *libre* quand elle n'est pas quantifiée. Une variable quantifiée est appelée *liée*.
- Attention : dans certains énoncés mal écrits, on peut trouver le même symbole pour une variable liée et une variable libre, par exemple dans $P(x, y) \wedge \forall x\, Q(x, y)$. Dans cet énoncé, x est-elle libre ou liée ?
- Pour éviter ce genre de problème il convient de renommer les variables liées en utilisant des symboles qui ne figurent pas parmi les variables libres. Cet énoncé devient donc $P(x, y) \wedge \forall z\, Q(z, y)$, dans lequel x et y sont libres et z est liée.
- Un énoncé sans variables libres est appelé *énoncé clos* ou *sentence*.

Théorie des modèles

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision



Aidez-moi à agir dans le monde !

Théorie des modèles / Structure, signature, interprétation

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Une *structure* est un triplet $\mathcal{A} = (\Delta, \sigma, \mathbf{I})$ où Δ est un *domaine* (ou *univers*), σ est une *signature* et $\mathbf{I}$ est une *fonction d'interprétation*.
- Un *domaine* Δ est un ensemble quelconque (non vide) dans lequel les variables vont prendre leurs valeurs ; il servira également à définir les images par la fonction d'interprétation des constantes, fonctions et prédicats.
- Une *signature* σ est un ensemble de symboles de fonction et de prédicat, ainsi qu'une fonction qui envoie ces symboles dans $\mathbb{N}_0$ appelée *arité*. Les fonctions d'arité 0 sont appelées des *constantes*.

Théorie des modèles / Structure, signature, interprétation

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Une *fonction d'interprétation* est une fonction $\mathbf{I}$ qui envoie :
  1. les constantes de σ dans Δ, on notera $\mathbf{I}(c) = c^{\mathcal{A}} \in \Delta$;
  2. toute fonction n-aire ($n \geq 1$) f de σ vers une fonction $\mathbf{I}(f) = f^{\mathcal{A}} : \Delta^n \to \Delta$;
  3. toute prédicat n-aire ($n \geq 1$) P de σ vers une fonction $\mathbf{I}(\mathrm{P}) = \mathrm{P}^{\mathcal{A}} : \Delta^n \to \{\bot, \top\}$, où $\bot$ signifie «faux» et $\top$ signifie «vrai».

Théorie des modèles / Interprétation

- Ayant défini $\mathbf{I}$ sur les éléments de σ on peut l'étendre aux termes : on obtient des applications $t^{\mathbf{I}}: \Delta^{j} \to \Delta$ pour chaque *terme* ayant j variables.
- De même, on peut l'étendre aux énoncés. On obtient donc une application $\alpha^{\mathbf{I}}: \Delta^{k} \to \{\top, \bot\}$ pour α en supposant que α a k variables libres.
- Autrement dit, si l'énoncé est *clos*, c'est-à-dire si il n'a aucune variable libre, alors toute interprétation lui associe une valeur de vérité (puisque $k = 0$).

Modèles, quantificateurs

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

Definition

Un *modèle* (= monde possible) est une *structure* qui rend un énoncé vrai.

Vision ensembliste de l'interprétation

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Si $\mathcal{A} = (\Delta, \sigma, \mathbf{I})$ est une structure, alors on peut associer à tout prédicat unaire P, le sous-ensemble $P^{\mathcal{A}}$ de Δ des $e \in \Delta$ tels que $P(e)$ soit vrai.
- Dans ce cas, si on dénote par $\top$ le prédicat toujours vrai, l'ensemble des $e \in \Delta$ pour lesquels $\top$ est vraie est tout Δ. De même, si $\bot$ est le prédicat toujours faux, c'est-à-dire l'ensemble des $e \in \Delta$ pour lesquels $\bot$ est vraie, cet ensemble est $\emptyset$.
- Si P et Q sont des prédicats, alors $(P \wedge Q)^{\mathcal{A}} = P^{\mathcal{A}} \cap Q^{\mathcal{A}}$ (pourquoi ?) et $(P \vee Q)^{\mathcal{A}} = P^{\mathcal{A}} \cup Q^{\mathcal{A}}$.
- Et $(\neg P)^{\mathcal{A}} = \Delta \setminus P^{\mathcal{A}}$.
- De même $(P \to Q)^{\mathcal{A}}$ est $(\neg P)^{\mathcal{A}} \cup Q^{\mathcal{A}} = (\Delta \setminus P^{\mathcal{A}}) \cup Q^{\mathcal{A}}$ et $(P \leftrightarrow Q)^{\mathcal{A}} = ((\Delta \setminus P^{\mathcal{A}}) \cup Q^{\mathcal{A}}) \cap ((\Delta \setminus Q^{\mathcal{A}}) \cup P^{\mathcal{A}}) = (\Delta \setminus (P^{\mathcal{A}} \cup Q^{\mathcal{A}})) \cup (P^{\mathcal{A}} \cap Q^{\mathcal{A}})$ (draw the patatoes !).

Vision ensembliste de l'interprétation

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Conclusion : l'interprétation d'une formule peut être considérée comme le résultat d'une série d'opérations ensemblistes sur des sous-ensembles du domaine. Si ce résultat est vide, la formule est insatisfaisable, s'il est Δ c'est une tautologie.

Tables de vérité

- On appelle *logique propositionnelle* le cas où on n'a ni prédicat (autre que les propositions), ni variable, ni fonction, ni quantificateur.
- En logique propositionnelle, on peut calculer la valeur vraie ou fausse d'un énoncé en appliquant les *tables de vérité* suivantes aux connecteurs :

A	$\neg A$
faux	vrai
vrai	faux

A	B	$A \wedge B$	$A \vee B$	$A \to B$	$A \leftrightarrow B$
faux	faux	faux	faux	vrai	vrai
faux	vrai	faux	vrai	vrai	faux
vrai	faux	faux	vrai	faux	faux
vrai	vrai	vrai	vrai	vrai	vrai

- (Ces tables nous montrent que l'on pourrait s'amuser à définir 2^4 opérateurs binaires, mais traditionnellement on se limite aux 4 ci-dessus.)

Tables de vérité

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Exemple : quelle est la valeur de $(P \wedge Q) \to (P \vee Q)$ selon celles de P et de Q ?

P	Q	$P \wedge Q$	$P \vee Q$	$(P \wedge Q) \to (P \vee Q)$
faux	faux	faux	faux	vrai
faux	vrai	faux	vrai	vrai
vrai	faux	faux	vrai	vrai
vrai	vrai	vrai	vrai	vrai

(On dira que cette formule est une *tautologie*.)

- On voit que dans ce cas bien particulier (celui de la logique propositionnelle) une interprétation n'est rien d'autre qu'une combinaison de valeurs des propositions (et donc une ligne de la table de vérité).

Interprétation

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

Definition

Un énoncé β est *conséquence* de α quand tout modèle de α est aussi modèle de β. On écrit alors $\alpha \models \beta$ (à ne pas confondre avec $\alpha \vdash \beta$ qui signifie que β peut être déduit de α par une suite d'inférences).

Theorem

Soient α et β deux formules. On a $\alpha \models \beta$ ssi $\alpha \to \beta$ est une tautologie (c'est-à-dire : l'implication est vraie pour toute interprétation).

Démonstration.

On va montrer que $\alpha \not\models \beta$ ssi $\alpha \to \beta$ n'est pas une tautologie. Ce dernier signifie qu'il existe $\mathcal{A}$ tel que $\mathcal{A} \models \neg(\alpha \to \beta)$, autrement dit que $\mathcal{A} \models (\alpha \wedge \neg\beta)$, autrement dit que $\mathcal{A} \models \alpha$ et $\mathcal{A} \not\models \beta$ ce qui est exactement dire que β n'est pas conséquence de α. □

Liens avec la langue naturelle

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Dès lors qu'elle a une valeur de vérité, une phrase peut interpréter une formule (ou inversement : une formule peut formaliser une phrase).
- Exemples : «Michel est élève de Télécom Bretagne», «je m'ennuie», «les Bretons n'aiment pas l'écotaxe».
- Contre-exemples : «700 millions de Chinois, et moi, et moi, et moi», «courage, fuyons !», «travailler plus pour gagner plus».
- Attention au comportement du connecteur $\to$:
«(Les coqs pondent des œufs) $\to$ (Télécom Bretagne est une grande école)» est vraie,
«(Les ânes volent) $\to$ (Télécom Bretagne est une discothèque)» est vraie aussi.

Usage des quantificateurs

- $\forall$ est le *quantificateur universel*, on l'utilise souvent suivi d'une implication :

tout homme est mortel

$$\forall x\,(\text{homme}(x) \to \text{mortel}(x)).$$

- Attention à ne pas confondre avec $\forall x\,(\text{homme}(x) \wedge \text{mortel}(x))$, qui a une toute autre signification (laquelle ?).
- $\exists$ est le *quantificateur existentiel*, on l'utilise souvent suivi d'une conjonction :

Il existe un élève de Télécom dont le prénom est Xavier

$$\exists x\,(\text{élèveTélécom}(x) \wedge \text{prénom}(x, \text{Xavier})).$$

- À ne pas confondre avec $\exists x\,(\text{élèveTélécom}(x) \to \text{prénom}(x, \text{Xavier}))$ (pourquoi ?).
- Exercice : comment interpréter $\forall x \exists y\, \text{aime}(x, y)$ et $\exists x \forall y\, \text{aime}(x, y)$?

Qu'est-ce que l'égalité ?

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- L'opérateur = dénote une relation spéciale, celle de l'*égalité* entre deux termes : $x = y$ signifie que x et y doivent être interprétés par la même valeur du domaine, quelque soit l'interprétation : $x^{\mathbf{I}} = y^{\mathbf{I}} = e \in \Delta$.
- De même, sa négation $\neq$ signifie que deux termes ne pourront jamais être interprétés par les mêmes valeurs du domaine.
- Pour dire, par exemple, que a a au moins deux frères, on écrira

$$\exists x\, \exists y\, (\text{frère}(x, a) \wedge \text{frère}(y, a) \wedge x \neq y)$$

dans le domaine des êtres humains (ou des nœuds d'un graphe, ou...).

QCDC : Question cruciale du cours

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- On se pose la question cruciale de ce cours :

Question cruciale de ce cours

Étant donnés une *base de connaissances* KB (c'est-à-dire la conjonction d'un ensemble d'énoncés vrais) et un énoncé α, est-ce qu'on a $\text{KB} \models \alpha$?

autrement dit : est-ce que tout modèle de KB est un modèle de α ? ou encore : est-ce que α est vrai dans tous les mondes possibles dans lesquels KB est vraie ? (et en particulier, le nôtre, si KB est une base de connaissances du monde réel).

QCDC : Question cruciale du cours

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Dans le cas de la logique propositionnelle, les modèles ne sont que des combinaisons de valeurs des propositions ; un algorithme (de «vérification de modèles») peut alors procéder par la force brute et vérifier cette conséquence pour toute combinaison de valeurs de vérité des propositions de α. Il sera de complexité exponentielle $O(2^n)$ où n est le nombre de propositions.
- Ceci ne s'applique plus quand on a des prédicats, fonctions et variables sur des domaines quelconques, potentiellement très grands ou infinis. Dans la suite on va découvrir d'autres méthodes pour répondre à la QCDC (= question cruciale du cours).

Exemple

- L'exemple classique est celui des *nombres naturels*.
- Comment définir $(\mathbb{N}, +)$? Il suffit de disposer d'un prédicat unaire NombreNaturel(), d'une constante 0 et d'un symbole de fonction $S()$ (= *successeur*) :

$$\begin{array}{l} \mathrm{NombreNaturel}(0) \\ \forall n\, \mathrm{NombreNaturel}(n) \rightarrow \mathrm{NombreNaturel}(S(n)) \\ \forall m\, \forall n,\, m \neq n \rightarrow S(m) \neq S(n) \text{ (sinon ?)} \\ \forall n\, 0 \neq S(n) \text{ (pourquoi ?)} \end{array}$$

- Ces axiomes sont appelés *axiomes de Peano*.

Les nombres naturels, suite

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- L'*addition* se définit alors de la manière suivante :

$$\begin{array}{l} \forall m\, \mathrm{NombreNaturel}(m) \rightarrow \mathrm{Addition}(0, m) = m \\ \forall m\, \forall n,\, \mathrm{NombreNaturel}(m) \wedge \mathrm{NombreNaturel}(n) \\ \quad \rightarrow \mathrm{Addition}(S(m), n) = S(\mathrm{Addition}(m, n)) \end{array}$$

- Et à partir des nombres entiers on obtient les rationnels, les réels, les complexes. À partir de l'addition, la multiplication, la division, l'exponentielle, etc.
- Une machine à qui on apprend les assertions ci-dessus est donc capable de calculer, elle a «connaissance» de l'*arithmétique*.

Satisfaisabilité

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature

Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Un énoncé est une *tautologie* si toutes ses interprétations sont des modèles. Exemple : $A \vee \neg A$. On note $\models A$.
- Un énoncé est *satisfaisable* s'il possède au moins un modèle.
- Un énoncé est *insatisfaisable* (comme Mick Jagger) s'il ne possède aucun modèle. Exemple : $\alpha \wedge \neg\alpha$. On dit aussi que c'est une *contradiction*.
- La notion d'insatisfaisabilité est très importante parce que la QCDC «$\mathrm{KB} \models \alpha$» est équivalente au problème de montrer que «$\mathrm{KB} \wedge \neg\alpha$ est insatisfaisable ».
- Un ensemble d'énoncés fermé pour la relation de conséquence est appelé une *théorie*, et les énoncés, des *théorèmes*.

Équivalence entre déduction et conséquence

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Un théorème fondamental de la logique du 1er ordre dit que les deux approches : celle de la déduction $\alpha \vdash \beta$ (= on déduit β de α en faisant des inférences) et celle de la conséquence $\alpha \models \beta$ (= β est conséquence de α si tout modèle de α est modèle de β), sont équivalentes.
- Autrement dit, pour montrer que $\mathrm{KB} \models \alpha$, qui est la QCDC, il suffit de déduire α de KB par une suite d'inférences.
- Reste à trouver un système déductif qui soit facilement implémentable. Pour cela, nous allons étudier un certain nombre de méthodes pour normaliser et simplifier la formule, et ensuite nous allons appliquer la *méthode de résolution*.

Équivalences logiques

- On dit qu'on a *équivalence logique* entre deux énoncés s'ils possèdent exactement les mêmes modèles. Voici une liste d'équivalences qui nous seront bien utiles pour faire des calculs :

$$
\begin{array}{rcll}
\alpha\wedge\beta & \equiv & \beta\wedge\alpha & \textit{commutativité}\\
\alpha\vee\beta & \equiv & \beta\vee\alpha & \text{commutativité}\\
\alpha\wedge(\beta\wedge\gamma) & \equiv & (\alpha\wedge\beta)\wedge\gamma & \textit{associativité}\\
\alpha\vee(\beta\vee\gamma) & \equiv & (\alpha\vee\beta)\vee\gamma & \text{associativité}\\
\neg\neg\alpha & \equiv & \alpha & \textit{double négation}\\
\neg(\alpha\wedge\beta) & \equiv & \neg\alpha\vee\neg\beta & \textit{De Morgan}\\
\neg(\alpha\vee\beta) & \equiv & \neg\alpha\wedge\neg\beta & \text{De Morgan}\\
\alpha\wedge(\beta\vee\gamma) & \equiv & (\alpha\wedge\beta)\vee(\alpha\wedge\gamma) & \textit{distributivité}\\
\alpha\vee(\beta\wedge\gamma) & \equiv & (\alpha\vee\beta)\wedge(\alpha\vee\gamma) & \text{distributivité}\\
\forall x\,\neg P & \equiv & \neg\exists x\,P & \\
\neg\forall x\,P & \equiv & \exists x\,\neg P & \\
\forall x\,P & \equiv & \neg\exists x\,\neg P & \\
\exists x\,P & \equiv & \neg\forall x\,\neg P &
\end{array}
$$

Un système déductif

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- On note $\textsc{Subst}(\{x/t\},\alpha)$ la *substitution* $\{x/t\}$, où l'on remplace la variable x par le terme t, appliquée à α.
- Par exemple $\textsc{Subst}(\{x/t\}, P(x)\wedge P(t))$ est $P(t)$.
- Une substitution remplace toujours une variable par un terme (= une variable, une constante, une fonction de termes).
- Attention : une variable ne doit pas être remplacée par un terme qui la contient, sinon c'est la boucle infinie…

Un système déductif

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- On a les deux règles d'inférence suivantes qui montrent tout l'intérêt de la substitution :

Proposition (Règle d'instantiation universelle)

$$\frac{\forall v\,\alpha}{\textsc{Subst}(\{v/g\},\alpha)}$$

où g est un terme.

- Plus intuitivement : si l'énoncé α est vrai pour tout le monde, alors elle est vraie pour un terme en particulier.

Un système déductif, suite

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

Proposition (Règle d'instantiation existentielle)

$$\frac{\exists v\,\alpha}{\textsc{Subst}(\{v/k\},\alpha)}$$

où k est une constante qui n'apparaît nulle part ailleurs dans KB.

- Plus intuitivement : si on nous dit qu'une valeur de v existe, alors autant lui donner un nom, même si on ne connaît pas explicitement sa valeur. Ce nom doit être nouveau !
- Exemple : $\exists x\in\mathbb{R}$ tel que $\frac{d(x^y)}{dy}=x^y$, on peut lui donner un nom (par exemple e) mais il ne faut pas que ce soit un nom déjà pris (par exemple π, i, etc.). On appelle ce genre de constante, une *constante de Skolem*.

Substitution

- Le processus de *substitution* est très important. Imaginons que l'on ait des énoncés atomiques $p_1, \ldots, p_n$ tels que $\bigwedge_i p_i \to q$, et que l'on ait des énoncés $p'_1, \ldots, p'_n$ provenant des p_i après une substitution ϑ : $p'_i = \textsc{Subst}(\vartheta, p_i)$ (le même ϑ pour tous les i). Alors la règle du *modus ponens généralisé* nous dit que :

$$\frac{p'_1, \ldots, p'_n \qquad p_1 \wedge \cdots \wedge p_n \to q}{\textsc{Subst}(\vartheta, q)} \textit{ modus ponens } \text{généralisé.}$$

- Exemple :

$$\frac{\text{humain(Socrate)} \qquad \forall x\,(\text{humain}(x) \to \text{mortel}(x))}{\text{mortel(Socrate)}} \textit{ m. p. } \text{gén}$$

- Ici $\vartheta = \{x/\text{Socrate}\}$, $p_1 = \text{humain}(x)$, $p'_1 = \text{humain(Socrate)}$, $q = \text{mortel}(x)$ et donc $\textsc{Subst}(\vartheta, q) = \text{mortel}(\textit{Socrate})$.

Unification

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- La substitution est utile seulement quand elle est effectuée dans le but de rendre les énoncés «similaires» et de permettre ainsi l'application du modus ponens généralisé. On appelle ce processus *unification* et la substitution obtenue, un *unificateur* :
  $\textsc{Unificateur}(p, q) = \vartheta$ tel que $\textsc{Subst}(\vartheta, p) = \textsc{Subst}(\vartheta, q)$.
- Exemple :
  $\textsc{Unificateur}(\text{humain}(x), \text{humain(Socrate)}) = \{x/\text{Socrate}\}$.

Exemple de (tentative d') unification

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- $\mathcal{L}_0 = (\mathrm{R}(f(g(x)), a, x), \mathrm{R}(f(g(a)), a, b), \mathrm{R}(f(y), a, z))$.
- y une variable, $g(a)$ est un terme ne contenant pas y, on fait $\text{sub}_1 = \{y/g(a)\}$:
- $\mathcal{L}_1 = (\mathrm{R}(f(g(x)), a, x), \mathrm{R}(f(g(a)), a, b), \mathrm{R}(f(g(a)), a, z))$.
- x une variable, a est un terme ne contenant pas x, on fait $\text{sub}_2 = \{x/a\}$:
- $\mathcal{L}_2 = (\mathrm{R}(f(g(a)), a, a), \mathrm{R}(f(g(a)), a, b), \mathrm{R}(f(g(a)), a, z))$.
- a et b sont des termes, on ne peut pas les substituer.
- Conclusion : $\mathcal{L}_0$ n'est pas unifiable.

Forme normale prénexe

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- On dit que l'énoncé φ est en *forme normale prénexe* (PNF) s'il est de la forme
$$\mathrm{Q}_1 x_1 \cdots \mathrm{Q}_n x_n \psi$$
où Q_i sont des quantificateurs et ψ est un énoncé sans quantificateurs.
- Exemples : $\forall x \exists y\, (f(x) = y)$ est en PNF. $\neg \forall x \exists y\, P(x, y, z)$ ne l'est pas (pourquoi ?), ni $\exists x \forall y\, \neg P(x, y, z) \wedge \forall x \exists y\, Q(x, y, z)$.

PNF

Theorem

Pour tout énoncé de logique du 1er ordre, il existe un énoncé en PNF qui lui est équivalent.

L'algorithme pour l'obtenir consiste à renommer les variables liées et à déplacer les quantificateurs de variables de manière à ce qu'ils englobent des parties de l'énoncé dans lesquelles ces variables n'apparaissent pas.

- Exemple d'application de l'algorithme PNF :

$$\begin{aligned}\varphi &\equiv \exists x \forall y\, \neg P(x,y,z) \land \forall x \exists y\, Q(x,y,z)\\ &\equiv \exists x \forall y\, \neg P(x,y,z) \land \forall u \exists v\, Q(u,v,z)\\ &\equiv \exists x \forall y \forall u \exists v\, (\neg P(x,y,z) \land Q(u,v,z)).\end{aligned}$$

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

CNF

- On dit qu'un énoncé est en CNF (*forme normale conjonctive*) s'il s'écrit sous forme d'une conjonction de disjonctions ($(F_1 \lor F_2) \land (G_1 \lor \neg G_2)$, etc.).

Proposition

Tout énoncé (sans quantificateurs) est équivalent à un énoncé en CNF.

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

Exemple

Voici l'algorithme pour l'obtenir :

1. remplacer les $\leftrightarrow$ par des conjonctions de $\rightarrow$;
2. remplacer les $\alpha \rightarrow \beta$ par des $\neg\alpha \lor \beta$;
3. faire entrer les négations dans les parenthèses pour obtenir des littéraux (autrement dit : appliquer les lois de *De Morgan*) ;
4. utiliser la propriété distributive.

Exemple :

Énoncé de départ : $B \leftrightarrow (F \lor G)$.
Élimination du $\leftrightarrow$: $B \rightarrow (F \lor G) \land (F \lor G) \rightarrow B$.
Élimination du $\rightarrow$: $(\neg B \lor F \lor G) \land (\neg(F \lor G) \lor B)$.
Faire entrer les $\neg$: $(\neg B \lor F \lor G) \land ((\neg F \land \neg G) \lor B)$.
Appliquer la *distributivité* : $(\neg B \lor F \lor G) \land (\neg F \lor B) \land (\neg G \lor B)$.

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

CPNF

- Quand un énoncé est sous forme prénexe et sa partie sans quantificateurs sous CNF, on dira qu'il est sous CPNF (*forme normale prénexe conjonctive*).

Theorem

Pour tout énoncé de logique du 1er ordre, il existe un énoncé en CPNF qui lui est équivalent.

Skolémisation

Definition

Un énoncé est sous *forme normale de Skolem* (SNF) s'il est en CPNF avec uniquement des quantificateurs universels.

- Il existe un algorithme qui fournit pour tout énoncé φ, un énoncé φ_S sous SNF que l'on appelle sa *skolémisation* :
  1. à partir de φ, obtenir $\varphi' = Q_1 \cdots Q_m\, \varphi(x_1, \ldots, x_m)$ en CPNF ;
  2. si tous les Q_i sont $\forall$, φ' est en SNF ;
  3. sinon, à partir de φ' obtenir $s(\varphi')$ qui a un $\exists$ en moins que φ' :
     – si le $\exists$ est en première position ($\exists x_1$), on supprime $\exists x_1$ et on remplace x_1 dans l'énoncé par une constante c qui n'apparaît pas dans φ' ;
     – si le $\exists$ est en i-ème position, on supprime $\exists x_i$ et on remplace x_i par une fonction $(i-1)$-aire $f(x_1, \ldots, x_{i-1})$ qui n'apparaît pas dans φ' ;
  4. en répétant cette étape k fois (pour k quant. exist.) on obtient $\overbrace{s \circ \cdots \circ s}^{k \text{ fois}}(\varphi')$, qui est en SNF.

Skolémisation

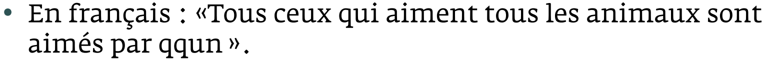
ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

Theorem

Soit φ un énoncé et φ^S son skolémisé. Alors φ est satisfaisable ssi φ^S est satisfaisable.

Exemple de skolémisation

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision



- En français : «Tous ceux qui aiment tous les animaux sont aimés par qqun ».
- En logique : $\forall x\, ((\forall y\, \mathrm{Animal}(y) \to \mathrm{Aime}(x,y)) \to (\exists y\, \mathrm{Aime}(y,x)))$;
- on élimine la 2e implication : $\forall x\, (\neg((\forall y\, \mathrm{Animal}(y) \to \mathrm{Aime}(x,y))) \vee (\exists y\, \mathrm{Aime}(y,x)))$;
- on élimine la 1re implication : $\forall x\, (\neg(\forall y\, (\neg(\mathrm{Animal}(y)) \vee \mathrm{Aime}(x,y))) \vee (\exists y\, \mathrm{Aime}(y,x)))$;
- on gère les négations : $\forall x\, (\exists y\, \neg(\neg \mathrm{Animal}(y) \vee \mathrm{Aime}(x,y))) \vee (\exists y\, \mathrm{Aime}(y,x)))$;
- et encore : $\forall x\, (\exists y\, (\mathrm{Animal}(y) \wedge \neg \mathrm{Aime}(x,y))) \vee (\exists y\, \mathrm{Aime}(y,x)))$;
- on skolemise : $\forall x\, (\mathrm{Animal}(f(x)) \wedge \neg \mathrm{Aime}(x, f(x))) \vee \mathrm{Aime}(g(x), x)$ où f et g sont des fonctions de Skolem ;
- on distribue la disjonction : $\forall x\, (\mathrm{Animal}(f(x)) \vee \mathrm{Aime}(g(x),x)) \wedge (\neg \mathrm{Aime}(x, f(x)) \vee \mathrm{Aime}(g(x),x))$.

Cet énoncé est plus opaque que celui du début, mais aussi plus maniable pour la machine. (Ici $f(x)$ est un «animal potentiellement non-aimé

Résolution

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- La *résolution* est une règle d'inférence facilement implémentable. Elle fonctionne sur des formules skolemisées.
- Soient deux disjonctions telles que la première contient un prédicat F et la deuxième la négation ¬F de celui-ci.
- Alors la résolution consiste à «éliminer» F et ¬F :

$$\frac{F \vee G \vee \neg H \qquad K \vee \neg F \vee L}{G \vee \neg H \vee K \vee L}\ \text{résolution}.$$

- La nouvelle disjonction obtenue est appelée *résolvant* des deux premières.
- En appliquant de manière répétée la règle de résolution, on obtient de plus en plus de résolvants.
- Le processus s'arrête en un temps fini puisque le nombre total de prédicats diminue de deux à chaque étape.

Résolution

Proposition

Si l'*ensemble vide* $\emptyset$ figure parmi les résolvants obtenus, alors l'énoncé de départ est une contradiction.

- Pour vérifier la QCDC (KB $\models \alpha$), on prend KB $\wedge \neg\alpha$, on le ré-écrit en SNF et on applique la règle de résolution à toutes les paires de disjonctions. À celles-ci viennent s'ajouter les résolvants et on continue à effectuer des résolutions jusqu'à ce qu'il n'y reste plus aucune disjonction non traitée. Le résultat obtenu va dépendre des choix de paires à résoudre, alors on recommence en faisant d'autres choix, etc. La formule KB $\wedge \neg\alpha$ est insatisfaisable si et seulement si au cours de ces opérations on a, à un moment donné, obtenu un résolvant vide.
- (Ce qui veut dire que si on veut montrer l'*insatisfaisabilité* on peut s'arrêter dès l'arrivée du $\emptyset$, par contre si on veut montrer la satisfaisabilité on doit traiter *toutes* les paires de clauses, clauses et résolvants, et résolvants entre eux.)

Résolution

- À noter que souvent pour faire une élimination on devra d'abord unifier les prédicats. Ainsi, par exemple, si la base de connaissances KB est

$$\begin{cases} \text{homme}(\text{Socrate}) \\ \forall X\, \text{homme}(X) \to \text{mortel}(X) \end{cases}$$

et α est « mortel(Socrate) », alors la résolution revient à montrer que la formule KB $\wedge \neg\alpha$, c'est-à-dire

$$\text{homme}(\text{Socrate}) \wedge (\neg\text{homme}(X) \vee \text{mortel}(X)) \wedge \neg\text{mortel}(\text{Socrate})$$

est insatisfaisable. Mais on voit tout de suite qu'on ne peut pas éliminer homme(Socrate) et $\neg$homme(X), ni mortel(X) et $\neg$mortel(Socrate).

Résolution

- C'est là que l'*unification* entre en jeu. On aura donc la résolution suivante :

$$\neg\text{homme}(X) \vee \text{mortel}(X)$$
$$\big|\ \{X/\text{Socrate}\}$$
$$\text{homme}(\text{Socrate}) \qquad \neg\text{homme}(\text{Socrate}) \vee \text{mortel}(\text{Socrate})$$
$$\neg\text{mortel}(\text{Socrate}) \qquad \text{mortel}(\text{Socrate})$$
$$\emptyset$$

Exemple de résolution

Soit la base de connaissances KB suivante :

$$\begin{cases} \text{pa}(\text{Robert}, \text{Kevin}) \\ \text{pa}(\text{Robert}, \text{Samantha}) \\ \text{pa}(\text{Samantha}, \text{Thomas}) \\ \text{pa}(\text{Doris}, \text{Jimmy}) \\ \text{pa}(\text{Boris}, \text{Jimmy}) \\ \text{pa}(\text{Boris}, \text{Elisabeth}) \\ \text{pa}(\text{Jimmy}, \text{Thomas}) \\ \text{pa}(\text{Samantha}, \text{Samuel}) \\ \text{pa}(\text{Jimmy}, \text{Samuel}) \\ \text{pa}(\text{Zelda}, \text{Max}) \\ \text{pa}(\text{Samuel}, \text{Max}) \\ \forall X, Y, Z\, \text{pa}(X, Z) \wedge \text{pa}(Z, Y) \to \text{gp}(X, Y) \end{cases}$$

où pa(X, Y) signifie que X est parent de Y, et gp(X, Y) que X est grand-parent de Y.

Exemple de résolution

- Soit α la formule suivante :
$$\exists X\, \mathrm{gp}(\mathrm{Robert}, X) \wedge \mathrm{pa}(X, \mathrm{Max}).$$
- On pose la QCDC : $\mathrm{KB} \models \alpha$? On sait quelle est équivalente à montrer que $\mathrm{KB} \wedge \neg\alpha$ est insatisfaisable.
- On prend donc $\mathrm{KB} \wedge \neg\alpha$ et on lui applique une résolution.
- À noter que comme $\neg\alpha$ est
$$\forall X\, \neg\mathrm{gp}(\mathrm{Robert}, X) \vee \neg\mathrm{pa}(X, \mathrm{Max}),$$
on a des quantificateurs universels partout, donc $\mathrm{KB} \wedge \neg\alpha$ est déjà skolemisée, on peut commencer à éliminer des prédicats dans les paires de disjonctions.

Exemple de résolution

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Allons-y :

$\mathrm{gp}(X, Y) \vee \neg\mathrm{pa}(X, Z) \vee \neg\mathrm{pa}(Z, Y)$ $\quad$ $\neg\mathrm{gp}(\mathrm{Robert}, X) \vee \neg\mathrm{pa}(X, \mathrm{Max})$

$\{X/\mathrm{Robert}\}$ $\quad$ $\{X/Y\}$

$\mathrm{gp}(\mathrm{Robert}, Y) \vee \neg\mathrm{pa}(\mathrm{Robert}, Z) \vee \neg\mathrm{pa}(Z, Y)$ $\quad$ $\neg\mathrm{gp}(\mathrm{Robert}, Y) \vee \neg\mathrm{pa}(Y, \mathrm{Max})$

$\neg\mathrm{pa}(\mathrm{Robert}, Z) \vee \neg\mathrm{pa}(Z, Y) \vee \neg\mathrm{pa}(Y, \mathrm{Max})$

- Ensuite il faut éliminer $\neg\mathrm{pa}(\mathrm{Robert}, Z)$. Un premier choix serait d'éliminer avec $\mathrm{pa}(\mathrm{Robert}, \mathrm{Kevin})$. Voici la résolution :

$\neg\mathrm{pa}(\mathrm{Robert}, Z) \vee \neg\mathrm{pa}(Z, Y) \vee \neg\mathrm{pa}(Y, \mathrm{Max})$

$\{Z/\mathrm{Kevin}\}$

$\mathrm{pa}(\mathrm{Robert}, \mathrm{Kevin})$ $\quad$ $\neg\mathrm{pa}(\mathrm{Robert}, \mathrm{Kevin}) \vee \neg\mathrm{pa}(\mathrm{Kevin}, Y) \vee \neg\mathrm{pa}(Y, \mathrm{Max})$

$\neg\mathrm{pa}(\mathrm{Kevin}, Y) \vee \neg\mathrm{pa}(Y, \mathrm{Max})$

Exemple de résolution

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Donc, pour avancer, la prochaine étape serait l'élimination de $\neg\mathrm{pa}(\mathrm{Kevin}, Y)$. Sauf que dans KB, il n'y a aucune formule avec la constante «Kevin» en première position. On s'aperçoit donc que l'on fait fausse route. Qu'à cela ne tienne ! On revient un pas en arrière et on essaie d'éliminer $\neg\mathrm{pa}(\mathrm{Robert}, Z)$ par le prédicat $\mathrm{pa}(\mathrm{Robert}, \mathrm{Samantha})$. On a donc :

$\neg\mathrm{pa}(\mathrm{Robert}, Z) \vee \neg\mathrm{pa}(Z, Y) \vee \neg\mathrm{pa}(Y, \mathrm{Max})$

$\{Z/\mathrm{Samantha}\}$

$\mathrm{pa}(\mathrm{Robert}, \mathrm{Samantha})$ $\quad$ $\neg\mathrm{pa}(\mathrm{Robert}, \mathrm{Samantha}) \vee \neg\mathrm{pa}(\mathrm{Samantha}, Y) \vee \neg\mathrm{pa}(Y, M$

$\neg\mathrm{pa}(\mathrm{Samantha}, Y) \vee \neg\mathrm{pa}(Y, \mathrm{Max})$

Exemple de résolution

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Cette fois-ci on a plus de possibilités pour éliminer Samantha : il y a Thomas et Samuel. Comme cela a été le cas avec Kevin, Thomas va également s'avérer être un mauvais choix, et c'est avec Samuel que l'on ira jusqu'au bout :

$\neg\mathrm{pa}(\mathrm{Samantha}, Y) \vee \neg\mathrm{pa}(Y, \mathrm{Max})$

$\{Y/\mathrm{Samuel}\}$

$\mathrm{pa}(\mathrm{Samantha}, \mathrm{Samuel})$ $\quad$ $\neg\mathrm{pa}(\mathrm{Samantha}, \mathrm{Samuel}) \vee \neg\mathrm{pa}(\mathrm{Samuel}, \mathrm{Max})$

$\mathrm{pa}(\mathrm{Samuel}, \mathrm{Max})$ $\quad$ $\neg\mathrm{pa}(\mathrm{Samuel}, \mathrm{Max})$

$\emptyset$

et donc $\mathrm{KB} \wedge \neg\alpha$ est insatisfaisable, et donc α est conséquence de KB, la réponse à la CQDC est affirmative.
- Notons que les substitutions effectuées lors de cette résolution nous fournissent aussi le renseignement à la question auxiliaire (mais tout aussi importante) : «finalement, qui est le X tel que $\mathrm{gp}(\mathrm{Robert}, X) \wedge \mathrm{pa}(X, \mathrm{Max})$?»

Exercice sur les

所有强龙都能喷火。
Aucun dragon fort ne peut ne pas souffler le feu.

一只狡猾的龙总是有角的。
Un dragon rusé a toujours des cornes.

弱龙是没有角的。
Aucun dragon faible n'a des cornes.

游客们只狩猎那些不喷火的龙。
Les touristes ne chassent que les dragons ne soufflant pas le feu.

狡猾的龙会让游客们感到害怕吗？即：它会被捉吗？
Un dragon rusé doit-il craindre les touristes ? (autrement dit : est-il chassé ?)

Monty Python : brûlez la sorcière !

ELU 610 Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- Dans leur film *Monty Python : Sacré Graal*, les Monty Python utilisent une logique un peu particulière pour prouver qu'une — au demeurant, très charmante — jeune femme (jouée par Connie Booth) est une sorcière.
- Nous allons montrer, en nous servant de la méthode de résolution, que leur raisonnement est fallacieux.

`http://www.youtube.com/watch?v=cMvZFRTih6g&t=1m11s`

Monty Python : brûlez la sorcière !

ELU 610 Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- (1) Que fait-on avec les sorcières ? On les brûle !
- (2) Que brûle-t-on d'autre ? Le bois !
- (3) Que fait le bois ? Il flotte.
- (4) Qui d'autre flotte ? Un canard.
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

ELU 610 Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- (1) $\forall X, \mathrm{Wi}(X) \to \mathrm{B}(X)$
- (2) Que brûle-t-on d'autre ? Le bois !
- (3) Que fait le bois ? Il flotte.
- (4) Qui d'autre flotte ? Un canard.
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X, \mathrm{Wi}(X) \to \mathrm{B}(X)$
- (2) $\forall X, \mathrm{Wo}(X) \to \mathrm{B}(X)$
- (3) Que fait le bois ? Il flotte.
- (4) Qui d'autre flotte ? Un canard.
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- (1) $\forall X, \mathrm{Wi}(X) \to \mathrm{B}(X)$
- (2) $\forall X, \mathrm{Wo}(X) \to \mathrm{B}(X)$
- (3) $\forall X, \mathrm{Wo}(X) \to \mathrm{F}(X)$
- (4) Qui d'autre flotte ? Un canard.
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- (1) $\forall X, \mathrm{Wi}(X) \to \mathrm{B}(X)$
- (2) $\forall X, \mathrm{Wo}(X) \to \mathrm{B}(X)$
- (3) $\forall X, \mathrm{Wo}(X) \to \mathrm{F}(X)$
- (4) $\mathrm{F}(D)$
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Rappel : signature
Système formel
Grammaire BNF de la logique du 1er ordre
Système déductif
Axiomes
Règles d'inférence
Exemple d'application des règles d'inférence
Et maintenant, que fait-on ?
Connecteurs et un quantificateur supplémentaires
Variables libres et liées
Théorie des modèles
Structure, signature, interprétation
Parenthèse : vision

- (1) $\forall X, \mathrm{Wi}(X) \to \mathrm{B}(X)$
- (2) $\forall X, \mathrm{Wo}(X) \to \mathrm{B}(X)$
- (3) $\forall X, \mathrm{Wo}(X) \to \mathrm{F}(X)$
- (4) $\mathrm{F}(D)$
- (5) $\forall X, Y, (\mathrm{F}(X) \wedge (w(X) = w(Y))) \to \mathrm{F}(Y)$
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X, \mathrm{Wi}(X) \rightarrow \mathrm{B}(X)$
- (2) $\forall X, \mathrm{Wo}(X) \rightarrow \mathrm{B}(X)$
- (3) $\forall X, \mathrm{Wo}(X) \rightarrow \mathrm{F}(X)$
- (4) $\mathrm{F}(\mathrm{D})$
- (5) $\forall X, Y, (\mathrm{F}(X) \wedge (w(X) = w(Y))) \rightarrow \mathrm{F}(Y)$
- (E) $w(\mathrm{JF}) = w(\mathrm{D})$
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X, \mathrm{Wi}(X) \rightarrow \mathrm{B}(X)$
- (2) $\forall X, \mathrm{Wo}(X) \rightarrow \mathrm{B}(X)$
- (3) $\forall X, \mathrm{Wo}(X) \rightarrow \mathrm{F}(X)$
- (4) $\mathrm{F}(\mathrm{D})$
- (5) $\forall X, Y, (\mathrm{F}(X) \wedge (w(X) = w(Y))) \rightarrow \mathrm{F}(Y)$
- (E) $w(\mathrm{JF}) = w(\mathrm{D})$
- (α) $\mathrm{Wi}(\mathrm{JF})$

Monty Python : brûlez la sorcière !

- (1) $\forall X, \neg\mathrm{Wi}(X) \vee \mathrm{B}(X)$
- (2) $\forall X, \neg\mathrm{Wo}(X) \vee \mathrm{B}(X)$
- (3) $\forall X, \neg\mathrm{Wo}(X) \vee \mathrm{F}(X)$
- (4) $\mathrm{F}(\mathrm{D})$
- (5) $\forall X, Y, \neg\mathrm{F}(X) \vee (w(X) \neq w(Y)) \vee \mathrm{F}(Y)$
- (E) $w(\mathrm{JF}) = w(\mathrm{D})$
- ($\neg\alpha$) $\neg\mathrm{Wi}(\mathrm{JF})$

Monty Python : brûlez la sorcière !

- (1) $\forall X, \neg\mathrm{Wi}(X) \vee \mathrm{B}(X)$
- (2) $\forall X, \neg\mathrm{Wo}(X) \vee \mathrm{B}(X)$
- (3) $\forall X, \neg\mathrm{Wo}(X) \vee \mathrm{F}(X)$
- (4) $\mathrm{F}(\mathrm{D})$
- (5) $\forall X, Y, \neg\mathrm{F}(X) \vee (w(X) \neq w(Y)) \vee \mathrm{F}(Y)$
- (E) $w(\mathrm{JF}) = w(\mathrm{D})$
- ($\neg\alpha$) $\neg\mathrm{Wi}(\mathrm{JF})$

Montrons que $(1) \wedge (2) \wedge (3) \wedge (4) \wedge (5) \wedge (E) \wedge (\neg\alpha)$ n'est pas insatisfaisable. Calculons tous les résolvants possibles :

- (4) et (5) : $\forall Y, (w(\mathrm{D}) \neq w(Y)) \vee \mathrm{F}(Y)$ (R1)
- (E) et (5) : $\neg\mathrm{F}(\mathrm{JF}) \vee \mathrm{F}(\mathrm{D})$ (R2)
- (R1) et (R2) : $(w(\mathrm{D}) \neq w(\mathrm{JF})) \vee \mathrm{F}(\mathrm{D})$ (R3)

Pas moyen d'avoir le $\emptyset$, donc $KB \not\models \alpha$.

Jesse Hoey : le *modus bogus* ☺

- Sur un **registre humoristique**, Jesse Hoey introduit la règle d'inférence

$$\frac{\alpha \to \beta \qquad \beta}{\alpha}\ \textit{modus bogus}.$$

- Il montre qu'avec cette règle d'inférence supplémentaire, le raisonnement des Monty Python est correct, et que la jeune fille est bel et bien une sorcière ! Voici comment il procède :
- Théorème. — $\forall X$, Wo(X) → Wi(X) (= *tout ce qui est de bois, est une sorcière*). *Preuve :* supposons Wo(X) pour un X donné. Alors par un *modus ponens* avec (2), on obtient B(X) et par un *modus bogus* avec (1), on obtient Wi(X).
- Proposition 1. — F(JF) (= *la jeune fille flotte*). *Preuve :* (4) et (E) donnent F(D) ∧ (w(JF) = w(D)). Ceci, par un *modus ponens* avec (5) donne F(JF).
- Proposition 2. — Wo(JF) (= *la jeune fille est de bois*). *Preuve :* prop. 1 et (3) par un *modus bogus*, donnent Wo(JF).
- Enfin, en combinant la prop. 2 et le théorème par un *modus ponens*, on a bien Wi(JF), cqfd.

Quatrième partie IV

Logiques de description

Les bases

- Les *logiques de description* sont des variantes de la logique du 1er ordre qui forment des compromis sur deux tableaux :
  1. elles ajoutent des nouvelles notations pour améliorer l'expressivité du langage (par ex. la possibilité de dire qu'il existe exactement n éléments ayant telle propriété) ;
  2. elles sont moins puissantes que la logique du 1er ordre, ce qui les rend décidables.

Les bases

- On a trois types d'objets :
  1. des *individus* (ce qui correspond aux constantes de la logique du 1er ordre),
  2. des *concepts* ou *classes* (des prédicats unaires, dont l'interprétation ensembliste correspond à des ensembles d'individus),
  3. des *rôles* ou *propriétés* (des prédicats binaires sur les individus) ;

Les bases

- On définit trois types de formules :
  1. celles de l'*ABox*, «A» comme «assertion», qui décrivent des concepts et des rôles (prédicats unaires et binaires sur des constantes) ;
  2. celles de la TBox, «T» comme «terminologie», qui décrivent des relations entre concepts ;
  3. celles de la RBox, «R» comme «relation», qui décrivent une hiérarchie des rôles, des compositions de rôles et des relations entre rôles, comme l'exclusion mutuelle, la réflexivité, la symétrie, la transitivité, etc.

ABox

- Une *assertion de concept* est un prédicat unaire du type ÉlèveTélécom(xavier).
- Une *assertion de rôle* est un prédicat binaire Père(andré,mathilde).
- Dans les logiques de description on ne fait pas l'*hypothèse de l'unicité des noms* : deux individus de même nom peuvent avoir le même référent, on écrira cloclo $\approx$ claudeFrançois, et samsonFrançois $\not\approx$ claudeFrançois.

TBox

- Puisqu'on peut interpréter les *concepts* (prédicats unaires) comme des ensembles, on peut aussi utiliser des relations de théorie d'ensembles :
- Mère $\sqsubset$ Parent, qui équivaut à $\forall X\, \text{Mere}(X) \to \text{Parent}(X)$;
- Personne $\equiv$ Humain, qui équivaut à $\forall X\, \text{Personne}(X) \leftrightarrow \text{Humain}(X)$.

RBox

- La *hiérarchie des concepts* induit une *hiérarchie des rôles* :
- ParentDe $\sqsubset$ AncêtreDe, qui équivaut à $\forall X, Y\, \text{ParentDe}(X, Y) \to \text{AncêtreDe}(X, Y)$.
- Une relation binaire peut se combiner avec une autre : sachant que le frère d'un père est un oncle, on peut écrire
- FrèreDe $\circ$ ParentDe $\sqsubset$ OncleDe, qui équivaut à $\forall X, Y, Z\, \text{FrereDe}(X, Y) \wedge \text{ParentDe}(Y, Z) \to \text{OncleDe}(X, Z)$.
- Enfin, on peut donner des caractéristiques générales des concepts : *Disjoint*(ParentDe, EnfantDe), ce qui équivaut à $\forall X, Y\, \text{ParentDe}(X, Y) \to \neg\text{Parent}(Y, X)$.

Restriction existentielle

- Toujours en mimant la théorie des ensembles, on peut écrire Mere $\equiv$ Femme $\sqcap$ Parent, qui équivaut à $\forall X\,\mathrm{Mere}(X) \leftrightarrow \mathrm{Femme}(X) \wedge \mathrm{Parent}(X)$,
- ou Parent $\equiv$ Pere $\sqcup$ Mere, qui équivaut à $\forall X\,\mathrm{Parent}(X) \leftrightarrow \mathrm{Mere}(X) \vee \mathrm{Pere}(X)$,
- la négation correspond au complémentaire d'un ensemble : $\neg$Célibataire est équivalent à Marié.
- L'ensemble complet correspond à un prédicat qui est toujours vrai, on l'écrit $\top$, pour exprimer une partition on écrira $\top \sqsubseteq$ Homme $\sqcup$ Femme.
- L'ensemble vide correspond à un prédicat qui est toujours faux, on l'écrit $\perp$, pour exprimer une exclusion mutuelle on écrira Homme $\sqcap$ Femme $\sqsubseteq \perp$.

Restriction existentielle

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Les bases
ABox
TBox
RBox
Constructeurs et restrictions
Constructeurs de concepts
Restrictions de rôle
Ontologies DL $\mathcal{SROIQ}$
Sémantique et vérification des langages de programmation

- Imaginons qu'on a un rôle $\mathrm{Parent}(X, Y)$ et que l'on cherche à caractériser les X qui sont parents. Il s'agit donc de dire «je cherche les X pour lesquels $\exists Y$ tel que $\mathrm{Parent}(X, Y)$», on écrira $\exists\mathrm{Parent}.\top$;
- le $\top$ signifie qu'on prend les X pour lesquels il existe un Y, sans les filtrer davantage.
- Si je cherchais ceux qui sont des parents d'au moins une fille, j'écrirais $\exists\mathrm{Parent}.\mathrm{Fille}$.
- De même, ceux qui n'ont que des filles : $\forall\mathrm{Parent}.\mathrm{Fille}$.
- Mais que signifie alors $\forall\mathrm{Parent}.\top$?

Restriction existentielle

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Les bases
ABox
TBox
RBox
Constructeurs et restrictions
Constructeurs de concepts
Restrictions de rôle
Ontologies DL $\mathcal{SROIQ}$
Sémantique et vérification des langages de programmation

- Pour le savoir, prenons les définitions formelles des sémantiques de ces notations.
- Soit R un rôle, et $R^{\mathbf{I}}$ son interprétation. Rappelons que dans une interprétation ensembliste, $R^{\mathbf{I}}$ devient un ensemble de paires d'éléments $R^{\mathbf{I}} = \{(x^{\mathbf{I}}, y^{\mathbf{I}}), \ldots\}$ du domaine Δ. On dira que $y^{\mathbf{I}}$ est un R-*successeur* de $x^{\mathbf{I}}$ si la paire $(x^{\mathbf{I}}, y^{\mathbf{I}})$ appartient à $R^{\mathbf{I}}$.
- Soit C un concept (et donc $C^{\mathbf{I}}$ est un sous-ensemble de Δ). Alors l'interprétation de $\exists R.C$ est $\{x^{\mathbf{I}} \mid$ quelques successeurs de $x^{\mathbf{I}}$ sont dans $C^{\mathbf{I}}\}$.
- Et celle de $\forall R.C$ est $\{x^{\mathbf{I}} \mid$ tous les successeurs de $x^{\mathbf{I}}$ sont dans $C^{\mathbf{I}}\}$.
- Donc, quelle est l'interprétation de $\forall\mathrm{Parent}.\top$? C'est le domaine tout entier. Effectivement, si $x^{\mathbf{I}}$ a des successeurs alors ils sont forcément dans l'interprétation de $\top$ qui est le domaine Δ. Et si $x^{\mathbf{I}}$ n'a pas de successeur alors l'assertion «tous ses successeurs sont dans Δ» demeure vraie aussi.

Restrictions au-plus et au-moins, réflexivité

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)
Les bases
ABox
TBox
RBox
Constructeurs et restrictions
Constructeurs de concepts
Restrictions de rôle
Ontologies DL $\mathcal{SROIQ}$
Sémantique et vérification des langages de programmation

- De même que l'on peut demander au moins un successeur ou tous les successeurs, on peut aussi spécifier «au moins n successeurs» : $\geq nR.C$, ainsi qu'«au plus n successeurs» : $\leq nR.C$.
- Une autre notation permet de décrire l'ensemble des éléments pour lesquels un rôle R est réflexif : $\exists R.\mathit{Self}$.
- Une manière de décrire un concept est en donnant explicitement ses membres : au lieu de Parent(jacques,julie), on peut aussi écrire $\{\mathrm{jacques}\} \sqsubseteq \exists\mathrm{Parent}.\{\mathrm{julie}\}$.
- On note R^{-} le *rôle inverse* de R, par exemple Parent^{-} est équivalent à Enfant puisque $\forall X, Y\,\mathrm{Parent}(X, Y) \leftrightarrow \mathrm{Enfant}(Y, X)$.
- Enfin, on note U le *rôle universel*, c'est-à-dire celui qui associe chaque élément à chaque autre élément (y compris lui-même).
- Enfin, on note R^{+} la *clôture transitive* de R, c'est-à-dire le plus petit rôle transitif contenant R.

Ontologies DL $\mathcal{SROIQ}$

- Une *ontologie* est la formalisation d'un domaine de connaissances dans un langage de représentation de connaissances.
- Une *ontologie* DL est une ontologie représentée dans le langage d'une logique de description.
- $\mathcal{SROIQ}$ est une logique de description spécifique que nous allons décrire dans la suite.
- Si N_C est un concept quelconque, N_R un rôle quelconque et N_I un individu quelconque, alors on définit de manière itérative dans $\mathcal{SROIQ}$:
  1. une *expression de concept* **C** par **C** ::= N_C | (**C** ⊓ **C**) | **C** ⊔ **C**) | ¬**C** | ⊤ | ⊥ | ∃**R**.**C** | ∀**R**.**C** | ≥*n* **R**.**C** | ≤*n* **R**.**C** | ∃**R**.*Self* | $\{N_I\}$,
  2. une *expression de rôle* **R** par **R** ::= U | N_R | N_R^-.
- Les axiomes d'une ontologie $\mathcal{SROIQ}$ sont des types suivants :
  1. ABox : $\mathbf{C}(N_I)$, $\mathbf{R}(N_I, N_I)$, $N_I \approx N_I$, $N_I \not\approx N_I$,
  2. TBox : $\mathbf{C} \sqsubset \mathbf{C}$, $\mathbf{C} \equiv \mathbf{C}$,
  3. RBox : $\mathbf{R} \sqsubset \mathbf{R}$, $\mathbf{R} \equiv \mathbf{R}$, $\mathbf{R} \circ \mathbf{R} \sqsubset \mathbf{R}$, *Disjoint*(**R**, **R**).

Propriétés de $\mathcal{SROIQ}$

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Les bases
ABox
TBox
RBox
Constructeurs et restrictions
Constructeurs de concepts
Restrictions de rôle
Ontologies DL $\mathcal{SROIQ}$
Sémantique et vérification des langages de programmation

- Voici les propriétés de la logique de description $\mathcal{SROIQ}$:
  1. $\mathcal{S}$ (également appelé $\mathcal{ALCR}^+$) : ($\mathcal{AL}$) présence de noms de concepts et de rôles, de ⊤, du constructeur ⊓ (conjonction), du quantificateur universel, ($\mathcal{C}$) de la négation de concept, ($\mathcal{R}^+$) de la clôture transitive ;
  2. $\mathcal{R}$: inclusion de rôles, réflexivité, irréflexivité, exclusion de rôles ;
  3. $\mathcal{O}$: possibilité de décrire un concept extensionnellement ($\{N_{I,1}, \ldots, N_{I,n}\}$) ;
  4. $\mathcal{I}$: possibilité d'avoir des rôles inverses (R^-) ;
  5. $\mathcal{Q}$: possibilité d'avoir des restrictions pleinement quantifiées ($\leq n$, $\geq n$).
- D'autres logiques de description sont éventuellement dotées des propriétés suivantes :
  6. $\mathcal{H}$: hiérarchie des rôles $R_1 \sqsubset R_2$, $\mathcal{H}$ est plus faible que $\mathcal{R}$;
  7. $\mathcal{N}$: restrictions de cardinalité plus faibles que $\mathcal{Q}$.

OWL

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Les bases
ABox
TBox
RBox
Constructeurs et restrictions
Constructeurs de concepts
Restrictions de rôle
Ontologies DL $\mathcal{SROIQ}$
Sémantique et vérification des langages de programmation

- La norme OWL 2 du consortium WWW correspond à la logique $\mathcal{SROIQ}$, alors que la norme plus faible OWL-DL correspond à $\mathcal{SHOIN}$.
- Le logiciel *Protégé* d'édition d'ontologies est basé sur cette dernière.

Cinquième partie V

Sémantique des langages de programmation

Programmes, états

ELU 610 Langages et logique
Yannis Haralambous (IMT Atlantique)
Qu'est-ce qu'un énoncé de langage de programmation ?
Le prédicat caractéristique
Assertions, pré- et postconditions
Sémantique d'un langage de programmation
L'affectation
La composition
Les tests
Les boucles
Comportement de wp
Le système déductif de

- Dans la première partie du module, on vérifie la correction syntaxique d'un programme à l'aide de la grammaire formelle du langage de programmation dans lequel il est écrit. Nous allons maintenant nous intéresser à la correction sémantique d'un programme.
- Mais comment décrire la sémantique d'un programme ?
- Un programme écrit dans un langage de programmation peut être considéré comme une *fonction* qui transforme l'état de mémoire en un autre état.
- Exemple : si les *variables* $(\mathtt{x},\mathtt{y})$ ont les valeurs $(8,7)$ à l'état (initial) s, après l'exécution de $\mathtt{x} := 2 * \mathtt{y} + 1$, on aura un état (final) s' avec des valeurs $(15,7)$.
- Si un programme utilise n variables $(\mathtt{x}_1,\dots,\mathtt{x}_n)$, un *état* s est un n-uplet de valeurs $(x_1,\dots,x_n)$.

Prédicat caractéristique

- Soit U l'ensemble de tous les n-uplets dans un domaine donné, et $U' \subseteq U$. Le *prédicat caractéristique* $P_{U'}$ de U', est défini par
$$U' = \{(x_1,\dots,x_n) \in U \mid P_{U'}(x_1,\dots,x_n)\}.$$
- Exemple : le code $\mathtt{x} := 2 * \mathtt{y} + 1$ transforme les états de $\{(x,y) \mid \text{vrai}\}$ en des états de $\{(x,y) \mid x = 2y+1\}$. Et ceux de $\{(x,y) \mid (y \leq 3)\}$ en des états de $\{(x,y) \mid (x \leq 7) \wedge (y \leq 3)\}$. On dira que $\mathtt{x} := 2 * \mathtt{y} + 1$ transforme $(y \leq 3)$ en $(x \leq 7) \wedge (y \leq 3)$.
- La *sémantique d'un langage de programmation* est la manière dont chaque énoncé transforme un état initial en un état final.
- Une *assertion* est un triplet $\{p\}\ \mathtt{S}\ \{q\}$ où $\mathtt{S}$ est un programme, et p et q des formules de la logique du 1er ordre appelées *précondition* et *postcondition*.

Assertions, correction partielle

- On dit qu'un programme *$\mathtt{S}$ est partiellement correct* par rapport à p et q, si on a la condition suivante : *quand $\mathtt{S}$ se termine après avoir commencé dans un état satisfaisant p, alors on se trouve dans un état satisfaisant q.*
- De même, on dit qu'un programme *$\mathtt{S}$ est totalement correct* par rapport à p et q, si on a la condition suivante : *quand $\mathtt{S}$ commence dans un état satisfaisant p, alors il se termine forcément après un nombre fini d'étapes, et on se trouve dans un état satisfaisant q.*
- Exemples : $\{y \leq 3\}\ \mathtt{x} := 2 * \mathtt{y} + 1\ \{(x \leq 7) \wedge (y \leq 3)\}$ est t.c., $\{\text{faux}\}\ \mathtt{S}\ \{q\}$ pour tout $\mathtt{S}$ et q est p.c., $\{p\}\ \mathtt{S}\ \{\text{vrai}\}$ pour tout $\mathtt{S}$ et q est p.c.

Ordre partiel des formules

- On dit qu'une formule B est *plus faible* qu'une formule A si $A \to B$.
- Exemples : $y = 1 \vee y = 3$ est plus faible que $y = 1$, $y \leq 3$ est plus faible que $y = 1 \vee y = 3$, $y \leq 4$ est plus faible que $y \leq 3$, etc.

Definition

Soit $\mathtt{S}$ un programme et q une formule, on note $\mathrm{wp}(\mathtt{S}, q)$ la *précondition la plus faible* telle que $\{\mathrm{wp}(\mathtt{S}, q)\}\ \mathtt{S}\ \{q\}$.

- Ainsi, pour tout p tel que $\{p\}\ \mathtt{S}\ \{q\}$, on aura $p \to \mathrm{wp}(\mathtt{S}, q)$.
- Intuitivement on peut dire que l'on cherche la précondition la plus faible parce que son prédicat aura le plus grand nombre de cas pour lesquels notre programme sera correct (= donnera le résultat escompté).

Sémantique d'un langage

- On peut considérer wp comme un *transformateur de prédicat*.
- D'où une autre manière de formaliser un programme : comme *une transformation d'un prédicat de postcondition en un prédicat de précondition*. (Approche axée sur les objectifs.)
- Pour définir la sémantique d'un langage donné, il suffit de la définir pour les instructions de base, et pour la manière de combiner les instructions (= la composition).
- Nous allons définir la sémantique de :
  1. l'affectation
  2. la composition d'instructions
  3. des tests `if`
  4. des boucles `while`.

L'affectation

- On définit l'instruction d'*affectation* `x := t` de la manière suivante :
$$\mathrm{wp}(\texttt{x := t}, p) = \textsc{Subst}(\{x/t\}, p)$$
pour tout p.
- Explication : on veut que, en conclusion, le prédicat p soit vrai. Après l'opération `x := t`, la seule valeur que x peut prendre est t. Donc on voit ce que ça donne quand on substitue x par t dans p. Voici quelques exemples :
  - $\{3 < 4 \wedge y \geq 5\}$ `x := 3` $\{x < 4 \wedge y \geq 5\}$ (on veut qu'à la fin $x < 4$; en faisant `x:=3`, ceci est toujours vrai (ce qui est le sens du prédicat $3 < 4$), donc il ne reste que la condition sur y) ;
  - $\{3 < 2 \wedge y \geq 5\}$ `x := 3` $\{x < 2 \wedge y \geq 5\}$ (on veut qu'à la fin $x < 2$; en faisant `x:=3`, ceci est toujours faux (ce qui est le sens du prédicat $3 < 2$), donc le programme ne marchera jamais) ;
  - $\{y = -1\}$ `x := 3` $\{2x + y = 5\}$ (on veut qu'à la fin $2x + y = 5$; en faisant `x:=3`, ceci n'est vrai que quand $2 \cdot 3 + y = 5$, c'est-à-dire quand $y = -1$.

La composition

- Gérer la *composition* de deux instructions :
$$\mathrm{wp}(\texttt{S}_1 ; \texttt{S}_2, q) = \mathrm{wp}(\texttt{S}_1, \mathrm{wp}(\texttt{S}_2, q))$$
pour tout q.
- Explication : pour arriver à q on passe par $\texttt{S}_1$ et ensuite par $\texttt{S}_2$. L'état intermédiaire entre ces deux opérations est la précondition la plus faible pour arriver à q à travers $\texttt{S}_2$, donc $\mathrm{wp}(\texttt{S}_2, q)$. On l'utilise en tant que postcondition de $\texttt{S}_1$, ce qui nous donne la formule.
- Exemple :
$$\begin{aligned}\mathrm{wp}(\texttt{x := x + 1; y := y + 2}, x < y) &= \mathrm{wp}(\texttt{x := x + 1}, \mathrm{wp}(\texttt{y := y + 2}, x < y))\\ &= \mathrm{wp}(\texttt{x := x + 1}, x < y + 2)\\ &= (x + 1 < y + 2) = (x < y + 1).\end{aligned}$$

Les tests

- On définit l'opération de *test*
$$\texttt{if B then S}_1 \texttt{ else S}_2$$
de la manière suivante :
$$\mathrm{wp}(\texttt{if B then S}_1 \texttt{ else S}_2, q) = (\texttt{B} \wedge \mathrm{wp}(\texttt{S}_1, q)) \vee (\neg\texttt{B} \wedge \mathrm{wp}(\texttt{S}_2, q))$$
pour tout q.
- Explication : pour arriver à la postcondition q il faut avoir soit B et la précondition de $\texttt{S}_1$, soit $\neg B$ et la précondition de $\texttt{S}_2$.
- Exemple : en PC.

Les boucles

- On définit l'opération de *boucle* `while B do S` de la manière suivante :
  $\mathrm{wp}(\mathtt{while\ B\ do\ S}, q) = ((\neg B \wedge q) \vee (B \wedge \mathrm{wp}(\mathtt{S; while\ B\ do\ S}, q))$
  pour tout q.
- Explication : pour arriver à la postcondition q il faut avoir soit $\neg B$ et q (le $\neg B$ fait que l'on s'arrête, et le q fait que l'on a q puisque rien n'a changé), soit B et la précondition de la composition de `S` et du test.
- La définition est récursive : on va exécuter `S` aussi longtemps que nécessaire pour que B devienne faux.
- Pour éviter d'avoir une énorme disjonction de termes $\mathrm{wp}(\mathtt{S; S; \ldots; S; while\ B\ do\ S}, q)$ on s'intéressera au cas particulier du prédicat p_{inv} tel que $\{p_{\mathrm{inv}}\}\ \mathtt{S}\ \{p_{\mathrm{inv}}\}$. On appelle p_{inv} un *invariant de boucle*.
- Exemple : en PC.

Système déductif (logique de Hoare)

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Qu'est-ce qu'un énoncé de langage de programmation ?
Le prédicat caractéristique
Assertions, pré- et postconditions
Sémantique d'un langage de programmation
L'affectation
La composition
Les tests
Les boucles
Comportement de wp
Le système

- 1. *Axiomes de domaine* (toute formule vraie dans le(s) domaine(s) des variables).
- 2. *Axiome d'affectation*
  $$\{\textsc{Subst}(\{x/t\}, p(x))\}\ \mathtt{x := t}\ \{p(x)\}.$$
- 3. *Règle de composition*
  $$\frac{\{p\}\ \mathtt{S}_1\ \{q\} \qquad \{q\}\ \mathtt{S}_2\ \{r\}}{\{p\}\ \mathtt{S}_1; \mathtt{S}_2\ \{r\}}.$$
- 4. *Règle de conséquence*
  $$\frac{p_1 \to p \qquad \{p\}\ \mathtt{S}\ \{q\} \qquad q \to q_1}{\{p_1\}\ \mathtt{S}\ \{q_1\}}.$$

Système déductif (logique de Hoare)

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Qu'est-ce qu'un énoncé de langage de programmation ?
Le prédicat caractéristique
Assertions, pré- et postconditions
Sémantique d'un langage de programmation
L'affectation
La composition
Les tests
Les boucles
Comportement de wp
Le système

- 5. *Règle de test*
  $$\frac{\{p \wedge B\}\ \mathtt{S}_1\ \{q\} \qquad \{p \wedge \neg B\}\ \mathtt{S}_2\ \{q\}}{\{p\}\ \mathtt{if\ B\ then\ S_1\ else\ S_2}\ \{q\}}.$$
- 6. *Règle de boucle*
  $$\frac{\{p \wedge B\}\ \mathtt{S}\ \{p\}}{\{p\}\ \mathtt{while\ B\ do\ S}\ \{p \wedge \neg B\}}.$$
  Pour montrer que $\{p\}\ \mathtt{while\ B\ do\ S}\ \{q\}$, pour un certain q, on va chercher un *invariant de boucle* p_{inv} de S (c'est-à-dire tel que $\{p_{\mathrm{inv}}\}\ \mathtt{S}\ \{p_{\mathrm{inv}}\}$), on va montrer que $p \to p_{\mathrm{inv}}$, et que $(p_{\mathrm{inv}} \wedge \neg B) \to q$ est vraie : alors, on conclut par la règle de conséquence, que $\{p\}\ \mathtt{while\ B\ do\ S}\ \{q\}$.

Exemple de vérification de (petit) programme

ELU 610 *Langages et logique*
Yannis Haralambous (IMT Atlantique)
Qu'est-ce qu'un énoncé de langage de programmation ?
Le prédicat caractéristique
Assertions, pré- et postconditions
Sémantique d'un langage de programmation
L'affectation
La composition
Les tests
Les boucles
Comportement de wp
Le système déductif de

$$P = \begin{cases} \{\text{vrai}\} \\ \mathtt{x := 0;} \\ \{x = 0\} \\ \mathtt{y := b;} \\ \{x = 0 \wedge y = b\} \\ \mathtt{while\ y\ <>\ 0\ do} \\ \quad p_{\mathrm{inv}} = \{x = (b - y) \cdot a\} \\ \quad \mathtt{begin\ x := x + a; y := y - 1;\ end} \\ \{x = b \cdot a\} \end{cases}$$

Que fait ce programme ? Le vérifier :

Theorem

$\{\text{vrai}\}\ \mathtt{P}\ \{x = b \cdot a\}$.

Exemple

Qu'est-ce qu'un énoncé de langage de programmation ?
Le prédicat caractéristique
Assertions, pré- et postconditions
Sémantique d'un langage de programmation
L'affectation
La composition
Les tests
Les boucles
Comportement de wp
Le système déductif de

Démonstration.

- On commence par la fin : ici, on nous donne l'invariant de boucle p_{inv}, il faut
  1. vérifier que c'est bien un invariant de boucle : $\{x + a = (b - y)a + a\}$ `x := x + a` $\{x = (b - y)a + a\}$ `y := y - 1` $\{x = (b - y)a\}$;
  2. vérifier qu'il est plus faible que la précondition de la boucle : on a bien $\{(x = 0) \wedge (y = b)\} \rightarrow \{x = (b - y)a\}$ puisque pour $x = 0$ et $y = b$, p_{inv} s'écrit $0 = 0$ et est donc vrai ;
  3. vérifier que $p_{\text{inv}} \wedge B$, B étant le test de la boucle, est plus fort que la postcondition du programme : la négation de B est $\neg(y \neq 0)$, c'est-à-dire $y = 0$, en l'appliquant à p_{inv} on trouve $\{x = b \cdot a\}$ qui est la postcondition.
- Après ces trois vérifications, on sait que $\{x = 0 \wedge y = b\}$ est précondition de la boucle pour la postcodition voulue.
- On applique alors l'axiome d'affectation deux fois, et on trouve que la précondition globale du programme est $\{0 = 0\}$, c'est-à-dire qu'il est vrai sans condition sur x et y. □

Exercice

ELU 610
Langages et logique
Yannis Haralambous (IMT Atlantique)

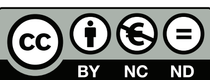
Qu'est-ce qu'un énoncé de langage de programmation ?
Le prédicat caractéristique
Assertions, pré- et postconditions
Sémantique d'un langage de programmation
L'affectation
La composition
Les tests
Les boucles
Comportement de wp
Le système déductif de

Vérifier la correction partielle du programme suivant :

$\{a > 0 \wedge b > 0\}$

```
x:=a;
y:=b;
while x<>y do
  begin
    if x>y
      then x:=x-y;
      else y:=y-x;
 end
```

$\{x = \text{PGCD}(a, b)\}$



ELU 610

PC de logique

Yannis Haralambous (IMT Atlantique)

1 Logique propositionnelle

1.1 Exercice

Lesquelles parmi les affirmations suivantes sont correctes ?

1. faux $\models$ vrai
2. vrai $\models$ faux
3. $(A \wedge B) \models (A \leftrightarrow B)$
4. $A \leftrightarrow B \models A \vee B$
5. $A \leftrightarrow B \models \neg A \vee B$
6. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B \vee C) \wedge (B \wedge C \wedge D \rightarrow E)$
7. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$
8. $(A \vee B) \wedge \neg(A \rightarrow B)$ est satisfaisable
9. $(A \wedge B) \rightarrow C \models (A \rightarrow C) \vee (B \rightarrow C)$
10. $(C \vee (\neg A \wedge \neg B)) \equiv ((A \rightarrow C) \wedge (B \rightarrow C))$
11. $(A \leftrightarrow B) \wedge (\neg A \vee B)$ est satisfaisable

1.2 Exercice

Décider lesquels parmi les énoncés suivants sont des tautologies, des contradictions ou ni l'un ni l'autre.

1. Fumée $\rightarrow$ fumée
2. Fumée $\rightarrow$ feu
3. (Fumée $\rightarrow$ feu) $\rightarrow$ ($\neg$fumée $\rightarrow$ $\neg$feu)
4. Fumée $\vee$ feu $\vee$ $\neg$feu
5. ((Fumée $\wedge$ chaleur) $\rightarrow$ feu) $\leftrightarrow$ ((fumée $\rightarrow$ feu) $\vee$ (chaleur $\rightarrow$ feu))
6. Grand $\vee$ stupide $\vee$ (grand $\rightarrow$ stupide)
7. (Grand $\wedge$ stupide) $\vee$ $\neg$stupide

1.3 Exercice

Considérer l'énoncé suivant :

$$((\text{nourriture} \rightarrow \text{boum}) \vee (\text{boissons} \rightarrow \text{boum})) \rightarrow ((\text{nourriture} \wedge \text{boissons}) \rightarrow \text{boum}).$$

1. Déterminer par la table de vérité si cet énoncé est une tautologie, s'il est satisfaisable ou insatisfaisable.
2. Convertir les deux côtés de l'implication CNF et en tirer des conclusions.
3. Prouver votre réponse à 1, en utilisant une résolution.

1.4 Exercice

Écrire sous forme d'énoncé de logique propositionnelle et vérifier :

1. Quand la temperature est constante et la pression est constante, il ne pleut pas. Donc si la temperature est constante et qu'il pleut, alors la pression a nécessairement varié.
2. Quand les gens ont faim, ils mangent. Quand Jean mange, il porte son meilleur constume. Aujourd'hui Jean n'a pas faim, donc il ne porte pas son meilleur costume.

2 Logique du 1er ordre

2.1 Exercice

Soit le vocabulaire suivant :

— $O(p, o)$ prédicat, la personne p occupe le poste p
— $C(p_1, p_2)$ prédicat, la personne p_1 est client de la personne p_2
— $B(p_1, p_2)$ prédicat, la personne p_1 est un supérieur hiérarchique (B comme «boss») de la personne p_2
— D, Ch, Av, Ac constantes, des métiers : docteur, chirurgien, avocat, acteur
— E, J constantes, des personnes : Émilie, Joël.

Utiliser ces symboles pour écrire les énoncés suivants en logique du 1er ordre :

1. Émilie est soit chirurgienne, soit avocate
2. Joël est acteur, mais il aussi un autre job
3. Tous les chirurgiens sont docteurs
4. Émilie a un boss qui est avocat
5. Il existe un avocat dont tous les clients sont des docteurs
6. Tout chirurgien a un avocat

2.2 Exercice

Soient les nombres naturels $\mathbb{N}$ avec le prédicat $<$, les fonctions $+$ et $\times$, et les constantes 0 et 1. Écrire en logique du 1er ordre :

1. la propriété de parité (Pair(x))
2. la propriété d'être nombre premier (Premier(x))
3. la conjecture de Goldbach [1].

2.3 Exercice

En utilisant les prédicats 1-aires ADN, Personne, Père, Mère et le prédicat 3-aire DérivéDe, écrire en logique du 1er ordre la phrase : l'ADN d'une personne est unique et dérivé de ceux de ses parents.

1. Tout nombre pair est somme de deux nombres premiers : $2 = 1 + 1$, $4 = 2 + 2$, $6 = 1 + 5$, $8 = 7 + 1$, $10 = 7 + 3$, $12 = 7 + 5$, $14 = 7 + 7$, ...

2.4 Exercice

Vrai ou faux ? Expliquer

1. $\exists x, x = \text{Rumpelstilzchen}$ est une tautologie.
2. Tout énoncé existentiellement quantifié est vrai dans tout modèle contenant exactement un objet.
3. $\forall x, y\ x = y$ est satisfaisable.

2.5 Affaire criminelle

Tous ceux qui aiment tous les animaux sont aimés par qqun
Quiconque tue un animal n'est aimé par personne
Jack aime tous les animaux
C'est soit Jack soit Curiosité qui a tué le chat appelé Luna
Montrer que c'est Curiosité qui a tué le chat

3 Inférence en logique du 1er ordre

3.1 Question

Comment montrer en utilisant la méthode de résolution qu'un énoncé α est une tautologie ? Une contradiction ?

3.2 Questions

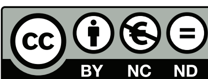
Soit une KB ne contenant que le seul énoncé $\exists x, \text{AussiHautQue}(x, \text{Everest})$. Lesquels parmi les énoncés suivants peuvent être dérivés de KB en appliquant une instantiation existentielle ?

1. $\text{AussiHautQue}(\text{Everest}, \text{Everest})$.
2. $\text{AussiHautQue}(\text{Kilimandjaro}, \text{Everest})$.
3. $\text{AussiHautQue}(\text{Kilimandjaro}, \text{Everest}) \wedge \text{AussiHautQue}(\text{MontsD'Arrhée}, \text{Everest})$.

3.3 Exercice

Trouver l'unificateur le plus général (s'il existe) pour les paires d'énoncés suivantes :

1. $P(A, A, B)$, $P(x, y, z)$
2. $Q(y, G(A, B))$, $Q(G(x, x), y)$
3. $\text{PlusVieux}(\text{Père}(y), y)$, $\text{PlusVieux}(\text{Père}(x), \text{Jerry})$
4. $\text{Connaît}(\text{Père}(y), y)$, $\text{Connaît}(x, x)$.



ELU 610

TP sur le langage Prolog

Yannis Haralambous (IMT Atlantique)

1 Introduction au langage Prolog

1.1 Le langage Prolog

Ce texte est une présentation du langage Prolog..

1.1.1 Faits et requêtes

Le langage Prolog nous permet d'exprimer des *faits* :

```
president_etats_unis(trump).
identifiant_de_cette_uv(elu610).
```

On écrit ces faits dans un fichier d'extension `.pro` (par exemple : `toto.pro`) et on lance Prolog :

```
swipl
```

Normalement on doit voir l'invite `?-`. Attention : elle ne sert qu'aux requêtes. Pour charger les faits et les règles, on les écrit dans un fichier (d'extension `.pl` ou `.pro`) et on utilise la commande `consult` :

```
consult('toto.pro').
```

(ne pas oublier les apostrophes et le point à la fin !).

Une fois les faits stockés en mémoire, on peut poser des requêtes :

```
identifiant_de_cette_uv(elu501).
```

(le système répond : `false.`) ou

```
president_etats_unis(trump).
```

(le système répond (malheureusement) : `true.`)

Ici, `identifiant_de_cette_uv` est un prédicat de formule logique et `elu610` un « atome ». L'arité du prédicat `identifiant_de_cette_uv` est 1.

En Prolog, l'opérateur $\wedge$ s'écrit sous forme d'une virgule :

```
president_etats_unis(trump), identifiant_de_cette_uv(elu610).
```

donne : `true.`

1.1.2 Variables

Les *variables* de formule logique commencent toujours par une lettre majuscule : `X`, `UnePersonne`, `Chaîne`, etc.

Si l'on pose la requête

```
president_etats_unis(X).
```

le système va trouver toutes les valeurs de `X` pour lesquelles cette formule logique est vraie, et va les afficher :

```
X = trump.
```

Au fait, le système a appliqué l'algorithme d'unification et nous montre les substitutions utilisées.

Une variable peut être partagée par plusieurs prédicats de la même formule logique. Si on n'est pas intéressé par la valeur de la variable, on peut utiliser `_` (une variable « joker »).

1.1.3 Règles

Dans un fichier Prolog on peut également inclure des *règles*. Celles-ci sont toujours de la forme « A est vrai si l'on a B. », et l'opérateur correspondant est noté `:-`.

Exemple :

```
Breton(X) :- NomFinitParEc(X), AimeLesCrêpes(X).
```

(= ceux dont le nom finit par «-ec» et qui aiment les crêpes, sont Bretons ; ce qui n'exclut pas qu'il y ait d'autres Bretons, n'aimant pas les crêpes ou leur nom ne se finissant pas par «-ec».)

Exercice : généalogie d'une famille Soient les faits suivants (fichier `famille.pro`, disponible sous Moodle) :

```
père(antoine,josianne).
père(michel,yves).
père(michel,yvonne).
père(yannick,jonathan).
père(eric,corinne).
père(michel,yannick).
père(yannick,jean).
père(yves,jimmy).
père(antoine,julie).
mère(yvonne,julie).
mère(marie,yannick).
mère(anne,jimmy).
mère(julie,corinne).
mère(marie,yvonne).
mère(marie,yves).
mère(adrienne,jonathan).
mère(adrienne,jean).
mère(yvonne,josianne).
```

(`père`(a, b) signifie que a est père de b, etc.)

1) Tracer l'arbre généalogique de cette famille !

On peut rajouter une règle qui définisse la notion de « grand-mère » :

```
grand-mère(X,Z) :- mère(X,Y), père(Y,Z).
grand-mère(X,Z) :- mère(X,Y), mère(Y,Z).
```

(pourquoi faut-il deux règles ?)

Si l'on demande

```
grand-mère(X,Y).
```

on obtient toutes les paires « grand-mère, petit-enfant » :

```
X = yvonne,
Y = corinne ;
X = marie,
Y = jean ;
X = marie,
Y = jonathan ;
X = marie,
Y = jimmy ;
X = marie,
Y = julie ;
X = marie,
Y = josianne ;
false.
```

Ici on doit taper un point-virgule pour obtenir la paire suivante. Le point-virgule est l'équivalent de l'opérateur $\vee$.

2) Définir une notion de `parent` et ré-écrire la définition de `grand-mère`.

3) En se servant du prédicat `parent`, définir la notion de cousin germain.

(Pour éviter d'être cousin germain de soi-même, inclure la clause `X \== Y`.)

4) Définir la notion d'ancêtre.

5) Définir la notion de descendant à partir de celle d'ancêtre.

1.2 Unification, résolution

Pour répondre aux requêtes, Prolog compare la formule demandée avec celles stockées en mémoire. Cette comparaison se fait à travers l'*unification*. L'unification consiste à comparer deux formules et à faire des substitutions judicieuses des variables par des termes (= des combinaisons de prédicats et d'atomes) jusqu'à ce que (après substitution) les deux formules soient identiques.

Ainsi, pour que `f(g(X,h(X,b)),Z)` devienne identique à `f(g(a,Z),Y)` il suffit de faire les substitutions `X` $\leftarrow$ `a`, `Z` $\leftarrow$ `h(a,b)`, `Y` $\leftarrow$ `h(a,b)`.

On peut demander à Prolog de tenter une unification. Écrivons, par exemple :

```
f(g(X,h(X,b)),Z) = f(g(a,Z),Y).
```

Sans que Prolog ait la moindre connaissance préalable sur `f`, `g`, il répond :

```
X = a,
Z = h(a, b),
Y = h(a, b).
```

Prolog utilise la *SLD-résolution* (une résolution où l'on prend toujours la clause la plus à gauche). Voici un exemple qui nous permettra de suivre les résolvants consécutifs :

```
p(X) :- q(X), r(X).
q(a).
q(b).
r(b).
r(c).
```

et la requête sera : `p(X)`.

Écrire la formule logique correspondant à ce programme et à la requête et faire la résolution.

Le premier résolvant est `p(X)`. Le deuxième résolvant est `q(X), r(X)`. Il a deux termes, Prolog commence par celui de gauche. Il va maintenant parcourir la liste des faits jusqu'à en trouver un qui puisse s'unifier avec ce terme. Le premier candidat est `q(a)`.

Pour unifier `q(X)` et `q(a)`, il faut substituer `X` par `a`. On obtient donc le troisième résolvant `q(a), r(a)`, qui est faux puisque `r(a)` est faux.

Prolog revient sur ses pas (on appelle cela le *backtracking*) et choisit une autre unification : pour unifier `q(X)` et `q(b)`, il substitue `X` par `b`. Le nouveau troisième résolvant est `q(b), r(b)`, qui est vrai. Donc `p(X)` est vérifié par `X = b`.

1.3 Problèmes potentiels

Soit le programme suivant :

```
p(X) :- p(X), q(X).
p(a).
q(a).
```

Écrire la formule logique correspondant à ce programme et à la requête et faire la résolution.

En théorie tout se passe bien, et pourtant Prolog réagit à la requête `p(X)` par :

```
ERROR: Out of local stack
   Exception: (261,566) p(_G236) ? creep
   Exception: (261,565) p(_G236) ? creep
   Exception: (261,564) p(_G236) ? creep
   Exception: (261,563) p(_G236) ? creep
   Exception: (261,562) p(_G236) ?
```

(le programme attend de nous de taper 261 562 passages à la ligne).

Autrement dit : ça plante. Pourquoi ?

Comment y rémédier très simplement ?

1.4 Les listes

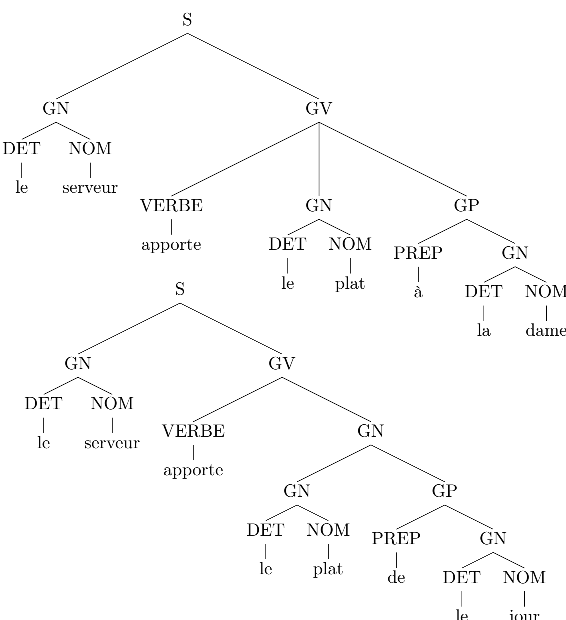
On note `[ ]` la liste vide, `[a, b, c, d]` une liste définie par énumération de ses objets, et `[a | X]` si `a` en est le premier élément et `X` *la liste de ce qui suit*, ou encore par exemple `[a, b, c | X]` si l'on distingue les trois premiers éléments.

Ainsi `[a | [b, c, d]]` = `[a, b | [c, d]]` = `[a, b, c | [d]]` = `[a, b, c, d | []]` = `[a, b, c, d]`. D'après Louis Gacôgne [5], *ce qui surprend les habitués d'un langage de programmation classique est qu'en Prolog, grâce au travail d'unification, il est possible d'écrire des expressions structurées. Ainsi pour les listes, le fait d'écrire une liste `[X | Q]` suffit pour que dans toute la clause où cette expression apparaît, `X` désigne la tête et `Q` la queue de cette liste qui n'a pas obligatoirement besoin d'être nommée.*

2 Prolog et langage naturel

Dans ce TP nous allons nous servir du langage Prolog pour définir des grammaires formelles, à l'aide desquelles nous analyserons des phrases (simples) de langage naturel, à l'occurrence : du français.

Comparons les deux phrases «le serveur apporte le plat à la dame» et «le serveur apporte le plat du jour». Dans le premier cas, «à la dame» est un COI et est donc rattaché au groupe verbal du verbe «apporte» (c'est la réponse à la question : «il l'apporte à qui ?»). Dans le deuxième cas, «du jour» est juste un complèment circonstanciel du COD «le plat», et est donc attaché à son groupe nominal.



où, bien sûr, «de le» se contracte en «du». Nous nous proposons d'écrire une grammaire formelle qui puisse analyser les deux types de phrase. Cette grammaire nous permettra de valider ou non une phrase française, en ce qui concerne la syntaxe, l'accord en genre et en nombre. Ensuite, nous nous pencherons sur sa sémantique.

[Rappels : GN est un groupe nominal, GV un groupe verbal, DET un déterminant, GP un groupe prépositionnel.]

2.1 Écrire des grammaires formelles en Prolog

Notre première règle sera

S
GN GV

autrement dit : une phrase qui comporte un groupe nominal et un groupe verbal, dans cet ordre. Si on écrit `s :- gn, gv.` en Prolog, on rate complètement l'ordre des deux groupes, puisque cette instruction n'est autre que la formule logique $\mathrm{gn} \wedge \mathrm{gv} \to \mathrm{s}$, et dans une conjonction l'ordre est indifférent.

Pour pallier ce problème, on se sert de l'astuce suivante : on considère que s, gn et gv sont des relations binaires et que l'on a

```
s(L1,L0) :- gn(L1,L2), gv(L2,L0).
```

où L_1, L_2 et L_0 sont des listes de mots. Les listes sont ordonnées, et donc l'ordre des groupes nominal et verbal est bien préservé. Une unification du contenu de chaque groupe est obtenue en prenant la différence des deux arguments : la phrase tout entière est $L_1 - L_0$, le groupe nominal est $L_1 - L_2$, le groupe verbal est $L_2 - L_0$ (la différence de deux listes $A = [a_1, \ldots, a_k, b_1, \ldots, b_m]$ et $B = [b_1, \ldots, b_m]$, est $A - B = [a_1, \ldots, a_k]$, donc $A \setminus B$ si les éléments de B se trouvent bien à la fin de A et dans le même ordre). Par exemple, on peut unifier les L_i de la manière suivante : L_1 est la phrase tout entière, L_2 est le groupe verbal et L_0 est vide.

Cette construction est tellement intéressante (et barbante à écrire) que Prolog définit une notation à part :


```
s --> gn, gv.
```

où la virgule n'a plus le même sens que dans les règles Prolog standard, puisque l'ordre de gn et de gv doit être respecté. Ceci n'est rien d'autre qu'une dérivation de grammaire formelle s→ gn gv, et c'est de cette manière que l'on décrit les grammaires formelles sous Prolog.

De même, l'écriture

```
verbe --> [mange].
```

signifie

```
verbe([mange | L], L).
```

Voici une grammaire qui permet d'analyser la très simple (et très poétique) phrase «la fille mange une pomme» :

```
s --> gn, gv.
gn --> [la, fille].
gn --> [une, pomme].
gv --> verbe, gn.
verbe --> [mange].
```

Attention : dans l'écriture `s --> gn, gv` *il s'agit bien de prédicats et non pas de constantes (les parenthèses sont «cachées»). Pour distinguer les constantes (qui seront les chaînes textuelles) on les met entre crochets (en utilisant ainsi la syntaxe de liste Prolog). Ainsi, dans* `verbe --> [mange]`*,* `verbe` *est en réalité le prédicat* `verbe(L,L')` *alors que* `mange` *est une constante.*

Pour la requête, Prolog n'a malheureusement pas prévu de notation spéciale. Pour vérifier qu'une phrase peut être générée par une grammaire d'axiome de départ `s`, on écrira la phrase sous forme de liste de mots, et on la fournira en tant que premier argument à la relation `s`. Le deuxième argument sera une liste vide :

```
?- s([la, fille, mange, une, pomme],[]).
true .
```

2.1.1 Exercice

Suivre à la trace le parcours de Prolog quand il répond à la requête ci-dessus.

En mettant une variable `X` en guise de premier argument de `s`, on obtient toutes les phrases valides pour cette grammaire : faites-le. Que constatez-vous ?

2.1.2 Exercice

Écrire une grammaire qui tient compte de déterminants («le», «la») et de prépositions («de», «à») et qui nous permette d'analyser les deux phrases «le serveur apporte le plat à la dame» et «le serveur apporte le plat de le jour». Vous pouvez vous inspirer des jolis arbres syntaxiques de la p. 1.

Tester la phrase «le serveur apporte le plat».

Tester la (mauvaise) phrase «le le». Que se passe-t-il ? Faites un `trace.` pour le découvrir.

2.2 Le problème de la récursion à gauche

Le problème rencontré provient de la règle `gn --> gn, gp.` Il est connu comme «problème de la récursion à gauche» (notons que, malgré les apparences, ceci ne relève pas du politique).

Si les règles

```
gn --> det, nom.
gn --> gn, gp.
```

posent un problème lorsqu'un groupe nominal n'est pas constitué d'un déterminant et d'un nom, intercaler un symbole auxiliaire qui ne soit pas récursif à gauche.

Que se passe-t-il maintenant lorsqu'on analyse `[le, le]` ?

2.3 Contracter «de» et «le» en «du»

Il y a bien des siècles que le français a contracté «de» et «le» en «du», il va falloir que notre système s'y mette. Ajouter des règles pour que la phrase «le serveur apporte le plat du jour» soit valide.

2.4 Accord en genre et en nombre

Ajouter le déterminant «les», la forme verbale «apportent», et les noms «plats», «serveurs», «dames» et «jours» au système.

Même si on vit dans un monde où l'orthographe est une espèce en voie d'extinction, ne permettons pas à notre système de valider des phrases comme «la serveur apporte les plat» ! S'il a ce défaut (partagé, hélas, par plusieurs de nos concitoyens) c'est parce que personne ne lui à appris à distinguer le masculin du féminin, le singulier du pluriel [1].

Pour ce faire, on va ajouter des variables `Genre` et `Nombre` aux noms et aux déterminants. De même on ajoutera une variable `Nombre` aux groupes nominaux et verbaux, et aux verbes. Mettre à jour le système pour qu'il ne se trompe plus dans les accords en genre et en nombre.

Attention : le verbe adopte le nombre du groupe nominal sujet (et non pas du groupe nominal complément d'objet). Vérifier les cas suivants :

1. «le serveur apporte le plat» : bon.
2. «le serveur apportent le plat» : mauvais.
3. «les serveur apporte le plat» : mauvais.
4. «les serveurs apporte le plat» : mauvais.
5. «les serveurs apportent le plat» : bon.
6. «les serveurs apportent les plat» : mauvais.
7. «les serveurs apportent les plats» : bon.
8. «le serveur apporte les plats» : bon.

(Tuyau : prévoir deux variables différentes pour le nombre du sujet et pour celui du complément d'objet.)

2.4.1 Exercice

Par quelle requête peut-on obtenir une liste de tous les noms masculins au pluriel ?

(Tuyau : on a vu que la notation `gn --> det, nom.` signifie en réalité `gn(L1,L0) :- det(L1,L2), nom(L2,L0).` Les variables supplémentaires s'ajoutent au début de la liste des arguments :

`gn(Genre,Nombre,L1,L0) :- det(Genre,Nombre,L1,L2), nom(Genre,Nombre,L2,L0).`)

2.5 Les verbes intransitifs

Ajouter les formes verbales «dort» et «dorment» au système.

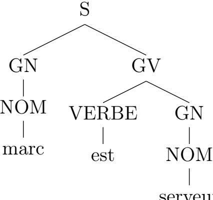
La phrase «le serveur dort» n'est pas validée par le système, et pourtant il arrive que les serveurs dorment. Pourquoi cette phrase n'est-elle pas couverte par notre analyse syntaxique ?

Modifier le système en ajoutant des clauses spécifiques aux verbes intransitifs `verbeintrans`.

2.6 Représentation sémantique, λ-calcul

Modulo l'ajout du nom «marc» et de la forme verbale «est», notre système va analyser la phrase «marc est serveur» tout bêtement comme :

1. Dans le cas de l'allemand, du russe, du grec, de l'arabe et de quelques autres langues, il faudrait également se poser le problème du cas (nominatif, génitif, datif, accusatif, vocatif, locatif, instrumental, etc.).



Cela nous informe sur la structure de la phrases mais ne nous (nous = humains, mais *aussi la machine !*) aide pas à comprendre la sémantique de la phrase. Ce que l'on pourrait extraire comme connaissance de cette phrase est : `est_serveur(marc)`, c'est-à-dire une relation (au sens de Prolog, et donc de la logique de premier ordre) `est_serveur` de la logique du premier ordre, avec comme argument `marc`. Cela correspond aussi à un triplet RDF avec comme sujet «marc» et comme prédicat «est serveur».

Mais comment obtenir, à partir d'un verbe quelconque, une relation (ou une fonction) logique qui puisse s'appliquer à un nom ? En mathématiques un raisonnement analogue serait : «à partir de l'écriture $f(x) = x^2$, je m'intéresse à f, et plus précisément à $x \mapsto x^2$» (cette écriture nous évite de donner un nom à la fonction, et nous dit clairement que la fonction n'a qu'une seule variable).

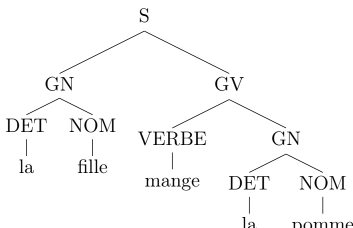
Le λ-calcul est un formalisme qui nous permet de définir cette opération, appelée λ-abstraction : à la place de $x \mapsto x^2$ on écrira $\lambda x.x^2$, le λx voulant dire que cette écriture comporte une variable liée, appelée x. Dans notre cas, ce serait $\lambda x.\texttt{est_serveur}(x)$ pour dénoter «la relation qui, appliquée à x, dit : "x est serveur"».

Imaginons maintenant un verbe transitif : «apporter». Ce verbe ne peut exister sans sujet, ni sans objet. On s'intéresse donc à $\lambda y.\lambda x.\texttt{apporte}(x, y)$ qui signifie «x apporte y». Puisque λy se trouve devant λx, on va d'abord appliquer cette relation à l'objet y, et ensuite à x. D'ailleurs, si on applique cette λ-expression à «le plat» on obtient $\lambda x.\texttt{apporte}(x, \text{le plat})$, c'est-à-dire la relation *à une variable liée* qui correspond à la sémantique «apporter le plat».

En Prolog on ne dispose pas de lettre λ, on utilisera, à la place, le caractère `^` : ainsi, `X^est_serveur(X)` signifie $\lambda x.\texttt{est_serveur}(x)$ et `Y^X^apporte(X,Y)` signifie $\lambda y.\lambda x.\texttt{apporte}(x, y)$.

2.6.1 Un exemple de grammaire avec représentation sémantique

Soit la phrase



Ici le verbe «manger» est transitif. On peut considérer le tableau suivant :

Nœud	Sémantique
GN (sous S)	la fille
GN (sous GV)	la pomme
VERBE	$\lambda y.\lambda x.\text{mange}(x, y)$
GV	$\lambda x.\text{mange}(x, \text{la pomme})$
S	mange(la fille, la pomme)

la dernière ligne duquel n'est autre que la sémantique de la phrase tout entière, écrite en Prolog. On peut s'inspirer de ce tableau pour écrire la grammaire suivante :

```
s(Semantique) --> gn(Sujet), gv(Sujet^Semantique).
gv(Sujet^Semantique) --> verbe(Objet^Sujet^Semantique), gn(Objet).
gn('la fille') --> ['la fille'].
gn('la pomme') --> ['la pomme'].
verbe(Y^X^mange(X,Y)) --> [mange].
```

Quelques explications. Essayons de comprendre ce qui se passe. Réécrivons ces cinq règles Prolog sans passer par l'opérateur `-->` :

```
s(L1,L0,Semantique) :- gn(L1,L2,Sujet), gv(L2,L0,Sujet^Semantique).
gv(L1,L0,Sujet^Semantique) :- verbe(L1,L2,Objet^Sujet^Semantique), gn(L2,L0,Objet).
gn(L1,L0,'la fille') :- L1=[la fille|L0].
gn(L1,L0,'la pomme') :- L1=[la pomme|L0].
verbe(L1,L0,'Y^X^mange(X,Y)') :- L1=[mange|L0].
```

Pour unifier `verbe(L1,L2,Objet^Sujet^Semantique)` et `verbe(L1,L0,'Y^X^mange(X,Y)')`, comme `Semantique` est une variable et `mange(X,Y)` un terme, on n'a pas d'autre choix que de procéder aux substitutions suivantes : `Objet/Y`, `Sujet/X`, `Semantique/mange(X,Y)`.

Mais dans ce cas, `gn(L1,L2,Sujet)` devient `gn(L1,L2,X)` et `gn(L2,L0,Objet)` devient `gn(L2,L0,Y)`. Après il n'y a plus qu'à substituer `X` et `Y` par `'la fille'` et `'la pomme'` et on obtient la substitution cumulée `Semantique/mange('la fille','la pomme')`.

Voici deux requêtes possibles :

```
?- s(Semantique, ['la fille',mange,'la pomme'],[]).
Semantique = mange('la fille', 'la pomme')

?- s(mange('la fille', 'la pomme'), L, []).
L = ['la fille', mange, 'la pomme']
```


Dans le premier cas, on donne la «forme de surface» et on récupère la sémantique. Dans le deuxième c'est l'inverse.

2.6.2 Exercice

Modifier la grammaire précédente pour tenir compte de l'éventuelle intransitivité du verbe «manger».

Quand on soumet `['la fille','mange']` à l'analyse, on obtient aussi la réponse suivante :

```
Semantique = _G322^mange(_G322, 'la fille') ;
```

De quoi s'agit-il ?

2.6.3 Imbrication

Le langage naturel permet d'imbriquer les sémantiques. Ainsi, la sémantique de la phrase «la fille qui a des yeux bleus mange la pomme» peut être représentée par :

```
mange(a('la fille','les yeux bleus'),'la pomme').
```

Écrire la grammaire qui produira cette représentation.

(Tuyaux : donner la structure logique d'une phrase subordonnée, le «qui» étant un symbole non-terminal spécifique de la grammaire. Une phrase subordonnée se comporte comme `s` sauf qu'elle est elle-même un `gn`. Prendre le cas plus simple où le sujet de la phrase subordonnée est forcement un nom.)

En admettant qu'une pomme peut avoir des yeux bleus, est-ce que cette grammaire validera aussi les phrases «la fille qui a les yeux bleus mange la pomme qui a les yeux bleus» et «la fille qui mange mange la pomme» ?

Et qu'en est-il de la phrase «la fille qui mange la pomme qui mange la pomme» ? Expliquer.

Et, enfin, qu'en est-il de «la fille qui mange la pomme qui mange la pomme mange la pomme» ?

2.6.4 Pour ceux qui veulent aller plus loin

Ce TP, inspiré de l'ouvrage [6], constitue une mini-introduction aux balbutiements du tout début d'une discipline appelée «sémantique computationnelle», qui sert, entre autres, à produire des outils de traitement automatique des langues. Dans ce TP, on n'a traité que les verbes par des λ-expressions. Les mêmes techniques s'appliquent aussi aux noms, aux adjectifs, aux déterminants, aux prépositions, bref : à toutes les parties du discours.

Notons que le langage naturel étant ambigu et imprévisible, les meilleurs outils de traitement de langue sont ceux qui combinent les techniques logiques présentées ici avec des techniques statistiques se basant sur l'apprentissage artificiel à partir de corpus de textes.

Pour une introduction tout à fait accessible et agréable à lire à la sémantique computationnelle, cf. [1]. Pour une description plus musclée du domaine, consulter [2, 3, 4].

Références

[1] P. Blackburn & J. Bos, *Representation and Inference for Natural Languages*, Center for the Study of Language and Information, 2005.

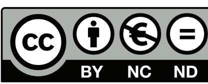
[2] H. Bunt & R. Muskens (ed.), *Computing Meaning*, vol. I, Springer, 1999.

[3] H. Bunt, R. Muskens & E. Thijsse (ed.), *Computing Meaning*, vol. II, Springer, 2001.

[4] H. Bunt & R. Muskens (ed.), *Computing Meaning*, vol. III, Springer, 2007.

[5] L. Gacôgne, *Programmation par l'exemple en Prolog*, Hermann, 2009.

[6] P. M. Nugues, *An Introduction to Language Processing with Perl and Prolog*, Springer, 2006.



INF 424

PC 3 de logique

Yannis Haralambous (Télécom Bretagne)

1 Sémantique et vérification de programmes

1.1 Exercice (invariant)

Montrer que $\{x = (b - y) \cdot a\}$ `x:=x+1; y:=y-1` $\{x = (b - y) \cdot a\}$.

1.2 Exercice (test)

Calculer wp(**if** $\mathtt{y} = 0$ **then** $\mathtt{x} := 0$ **else** $\mathtt{x} := \mathtt{y} + 1, x = y$).

1.3 Exercice (boucle)

Calculer wp(**while** $\mathtt{x} > 0$ **do** $\mathtt{x} := \mathtt{x} - 1, x = 0$).
Que se passe-t-il si on remplace $x = 0$ par $x = 1$?

1.4 Vérification de programme

Vérifier la correction partielle du programme suivant :

$\{a \geq 0\}$

```
x:=0; y:=1;
while y<=a do
begin
x:=x+1;
y:=y+2*x+1
end
```

$\{0 \leq x^2 \leq a < (x+1)^2\}$

1.5 Vérification de programme

Vérifier la correction partielle du programme suivant :

$\{a > 0 \wedge b > 0\}$

```
x:=a; y:=b;
while x<>y do
if x>y
then x:=x-y
else y:=y-x
```

$\{x = \mathrm{PGCD}(a, b)\}$

1.6 Vérification de programme

Vérifier la correction partielle du programme suivant :

$\{a \geq 0 \wedge b \geq 0\}$

```
x:=a; y:=b; z:=1
while y<>0 do
if impair(y)
then begin y:=y-1; z:=x*z end
else begin x:=x*x; y:=y div 2 end
```

$\{z = a^b\}$

IMT Atlantique Bretagne-Pays de la Loire
http://www.imt-atlantique.fr

Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 3
France

Campus de Nantes
4, rue Alfred Kastler - La Chantrerie
CS 20722
44307 Nantes Cedex 3
France

Campus de Rennes
2, rue de la Châtaigneraie
CS 17607
35576 Cesson-Sévigné Cedex
France

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom